利用Python进行数据分析

Jobrest

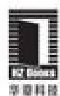
Published with GitBook



Table of Contents

利用Python进行数据分析 0

利用Python进行数据分析



Python for Data Analysis

利用Python
进行数据分析



O'REILLY®
机械工业出版社
China Machine Press

Wes McKinney 著
唐学韬 等译

Table of Contents

重要的Python库

安装和设置

社区和研讨会

使用本书

致谢

第2章　引言

来自bit.ly的1.usa.gov数据

MovieLens 1M数据集

1880—2010年间全美婴儿姓名

小结及展望

第3章　IPython：一种交互式计算和开发环境

IPython基础

内省

使用命令历史

与操作系统交互

软件开发工具

IPython HTML Notebook

利用IPython提高代码开发效率的几点提示

高级IPython功能

致谢

第4章　NumPy基础：数组和矢量计算

NumPy的ndarray：一种多维数组对象

通用函数：快速的元素级数组函数

利用数组进行数据处理

用于数组的文件输入输出

线性代数

随机数生成

范例：随机漫步

第5章　pandas入门

pandas的数据结构介绍

基本功能

汇总和计算描述统计

处理缺失数据

层次化索引

其他有关pandas的话题

第6章　数据加载、存储与文件格式

读写文本格式的数据

二进制数据格式

使用HTML和Web API

使用数据库

第7章　数据规整化：清理、转换、合并、重塑

合并数据集

重塑和轴向旋转

数据转换

字符串操作

示例：USDA食品数据库

第8章　绘图和可视化

matplotlib API入门

pandas中的绘图函数

绘制地图：图形化显示海地地震危机数据

Python图形化工具生态系统

第9章　数据聚合与分组运算

GroupBy技术

数据聚合

分组级运算和转换

透视表和交叉表

示例：2012联邦选举委员会数据库

第10章　时间序列

日期和时间数据类型及工具

时间序列基础

日期的范围、频率以及移动

时区处理

时期及其算术运算

重采样及频率转换

时间序列绘图

移动窗口函数

性能和内存使用方面的注意事项

第11章　金融和经济数据应用

数据规整化方面的话题

分组变换和分析

更多示例应用

第12章　NumPy高级应用

ndarray对象的内部机理

高级数组操作

广播

ufunc高级应用

结构化和记录式数组

更多有关排序的话题

NumPy的matrix类

高级数组输入输出

性能建议

附录A　Python语言精要

O'Reilly精品图书系列

利用Python进行数据分析

Python for Data Analysis

（美）麦金尼（McKinney,W.）　著

唐学韬　译

ISBN：978-7-111-43673-7

本书纸版由机械工业出版社于2014年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目　录

前言

第1章　准备工作

本书主要内容

为什么要使用Python进行数据分析

重要的Python库

安装和设置

社区和研讨会

使用本书

致谢

第2章　引言

来自bit.ly的1.usa.gov数据

MovieLens 1M数据集

1880—2010年间全美婴儿姓名

小结及展望

第3章　IPython：一种交互式计算和开发环境

IPython基础

内省

使用命令历史

与操作系统交互

软件开发工具

IPython HTML Notebook

利用IPython提高代码开发效率的几点提示

高级IPython功能

致谢

第4章　NumPy基础：数组和矢量计算

NumPy的ndarray：一种多维数组对象

通用函数：快速的元素级数组函数

利用数组进行数据处理

用于数组的文件输入输出

线性代数

随机数生成

范例：随机漫步

第5章　pandas入门

pandas的数据结构介绍

基本功能

汇总和计算描述统计

处理缺失数据

层次化索引

其他有关pandas的话题

第6章　数据加载、存储与文件格式

读写文本格式的数据

二进制数据格式

使用HTML和Web API

使用数据库

第7章　数据规整化：清理、转换、合并、重塑

合并数据集

重塑和轴向旋转

数据转换

字符串操作

示例：USDA食品数据库

第8章　绘图和可视化

matplotlib API入门

pandas中的绘图函数

绘制地图：图形化显示海地地震危机数据

Python图形化工具生态系统

第9章　数据聚合与分组运算

GroupBy技术

数据聚合

分组级运算和转换

透视表和交叉表

示例：2012联邦选举委员会数据库

第10章　时间序列

日期和时间数据类型及工具

时间序列基础

日期的范围、频率以及移动

时区处理

时期及其算术运算

重采样及频率转换

时间序列绘图

移动窗口函数

性能和内存使用方面的注意事项

第11章　金融和经济数据应用

数据规整化方面的话题

分组变换和分析

更多示例应用

第12章　NumPy高级应用

ndarray对象的内部机理

高级数组操作

广播

ufunc高级应用

结构化和记录式数组

更多有关排序的话题

NumPy的matrix类

高级数组输入输出

性能建议

附录A　Python语言精要

O'Reilly Media,Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

译者序

说句真心话，我非常感谢有机会翻译这本书，所以这可算是第一篇我自己真正想写的译者序。虽然之前也翻译过好几本书，但都没有这次的感悟这么多、这么深！这本书是我花精力和时间最多，同时也是最不满意的一本，就是因为这些感悟——我始终觉得，如果再多点时间的话，我还可以翻译得更好。

本书的内容非常好，至少有一点非常好——集中火力对付特定的应用领域。市面上介绍编程的书多如牛毛，但几乎没有几本书是针对特定应用场景的。这本书对新手来说绝对是福音，因为每看完一点就可以马上将自己手上的工作直接拿来当例子练手，这种立竿见影的学习效果，绝对会增强新手的学习信心。

本书内容虽好，但由于作者是编辑界牛人，平时的工作肯定不少，写书方面的精力自然就不可能太多。加之美式英语本来就很口语化，导致原书口水话非常多，有些地方的从句跟绕口令似的。我在翻译的过程中尽量排除了一些，两次校稿的过程中又删除或大幅修改了一些废话，虽然这种“口水话”还存在不少，但至少不会对阅读造成太大影响。如果实在觉得语言不通顺，请随时发邮件给我，欢迎大家的善意指导（tonytang1999@126.com）。

此外，在翻译的过程中发现了不少小问题，用词方面的错误几乎都是直接改的（小部分写了译者注，因为编辑要求我尽量标出一些来以便核对），而其他错误则几乎全部采用译者注的形式说明，还有一些原文有歧义或不详尽的地方也通过译者注的形式给出了简单说明。

本书共12章，除非你已经什么都会了，否则我建议全部阅读。如果没有学过Python，建议先看看本书后面的附录。本书所用到的Python编程基础知识很少，所以只看那个附录完全足够了。但是，如果你一点儿编程基础都没有的话，可能需要再看一本有关Python入门的书才行（比如《Python编程实践》编注1）。

对了，还有几件事情需要说明一下：

·每章的代码示例最好在一个IPython会话中完成，否则可能会出现一些不必要的麻烦，比如“xxx未定义”。

·如果在Windows里面用IPython，复制代码的时候建议使用cpaste，这个不多解释了。

·有关地图的那段代码可能需要找英文资料看才行，我在译者注中也说明了。这可能需要花不少时间和精力。

·由于原文各种说法不统一（甚至包括术语），虽然我尽量做了统一处理，但由于精力和时间有限，无法完全修改，所以译文中的“xxx接受yyy”、“将yyy传入xxx”说的都是“xxx函数有yyy这么个参数”；“选项”、“位置参数”、“关键字参数”、“形参”、“实参”说的都是“参数”……还有不少，我也记不清了。

·“金融和经济数据”那一章翻译得非常痛苦，因为我根本不了解那个行业，原文的术语又不标准，于是我基本都是用wikipedia和bing查英文资料，看懂之后再到baidu找中文资料，并最终确定译文。因此，可能会有不准确的情况，如果您发现了，请及时通过邮件告诉我，万分感谢。

此外，我必须感谢华章公司的编辑们。非常感谢他们能够给我这样的机会，也非常感谢他们在整个过程中给予我的各种支持和理解。希望以后还能有更加愉快的合作。

本书大部分内容的翻译工作以及全书的统稿工作由我完成，参与本书翻译校对工作的还有黄惠庄、卢彦良、蒲巧惠、陈丽丽、胡元江、张杨、赵杰、吴斌、郭敏、林丹、王跃等。

由于译者水平有限，书中肯定会存在一些错误或不妥之处，因此，在阅读过程中发现有任何问题，请随时联系我们（tonytang1999@126.com）或机械工业出版社，我们将及时更新本书的勘误表。当然，也非常欢迎大家对本书提出宝贵的意见和建议。

唐学韬

2013年6月于广州

编注1

:本书已由机械工业出版社出版，ISBN:978-7-111-36478-8。

前言

针对科学计算领域的Python开源库生态系统在过去10年中得到了飞速发展。2011年底，我深深地感觉到，由于缺乏集中的学习资源，刚刚接触数据分析和统计应用的Python程序员举步维艰。针对数据分析的关键项目（尤其是NumPy、matplotlib和pandas）已经很成熟了，也就是说，写一本专门介绍它们的图书貌似不会很快过时。因此，我下定决心要开始这样的一个写作项目。我在2007年刚开始用Python进行数据分析工作时就希望能够得到这样一本书。希望你也能觉得本书有用，同时也希望你能将书中介绍的那些工具高效地运用到实际工作中去。

本书的约定

本书使用了以下排版约定：

斜体（Italic）

用于新术语、URL、电子邮件地址、文件名与文件扩展名。

等宽字体（Constant width）

用于表明程序清单，以及在段落中引用的程序中的元素，如变量、函数名、数据库、数据类型、环境变量、语句、关键字等。

等宽粗体（Constant width bold）

用于表明命令，或者需要读者逐字输入的文本内容。

等宽斜体（Constant width italic）

用于表示需要使用用户提供的值或者由上下文决定的值来替代的文本内容。

注意： 代表一个技巧、建议或一般性说明。

警告： 代表一个警告或注意事项。

示例代码的使用

本书提供代码的目的是帮你快速完成工作。一般情况下，你可以在你的程序或文档中使用本书中的代码，而不必取得我们的许可，除非你想复制书中很大一部分代码。例如，你在编写程序时，用到了本书中的几个代码片段，这不必取得我们的许可。但若将O拎eilly图书中的代码制作成光盘并进行出售或传播，则需获得我们的许可。引用示例代码或书中内容来解答问题无需许可。将书中很大一部分的示例代码用于你个人的产品文档，这需要我们的许可。

如果你引用了本书的内容并标明版权归属声明，我们对此表示感谢，但这不是必需的。版权归属声明通常包括：标题、作者、出版社和ISBN号，例如："Python for Data Analysis by William Wesley McKinney (O'eilly).Copyright 2013William Wesley McKinney,978-1-449-31979-3"。

如果你认为你对示例代码的使用已经超出上述范围，或者你对是否需要获得示例代码的授权还不清楚，请随时联系我们：permissions@oreilly.com。

联系我们

有关本书的任何建议和疑问，可以通过下列方式与我们取得联系：

美国：

O'eilly Media,Inc.

1005Gravenstein Highway North

Sebastopol,CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

我们会在本书的网页中列出勘误表、示例和其他信息。可以通过http://oreil.ly/Python_for_Data_Analysis访问该页面。

要评论或询问本书的技术问题，请发送电子邮件到：

bookquestions@oreilly.com

想了解关于O'eilly图书、课程、会议和新闻的更多信息，请访问以下网站：

http://www.oreilly.com.cn

http://www.oreilly.com

还可以通过以下网站关注我们：

我们在Facebook上的主页：http://facebook.com/oreilly

我们在Twitter上的主页：http://twitter.com/oreillymedia

我们在YouTube上的主页：http://www.youtube.com/oreillymedia

第1章　准备工作

本书主要内容

本书讲的是利用Python进行数据控制、处理、整理、分析等方面的具体细节和基本要点。同时，它也是利用Python进行科学计算的实用指南（专门针对数据密集型应用）。本书重点介绍了用于高效解决各种数据分析问题的Python语言和库。本书没有阐述如何利用Python实现具体的分析方法。

当书中出现“数据”时，究竟指的是什么呢？主要指的是结构化数据（structured data），这个故意含糊其辞的术语代指了所有通用格式的数据，例如：

·多维数组（矩阵）。

·表格型数据，其中各列可能是不同的类型（字符串、数值、日期等）。比如保存在关系型数据库中或以制表符/逗号为分隔符的文本文件中的那些数据。

·通过关键列（对于SQL用户而言，就是主键和外键）相互联系的多个表。

·间隔平均或不平均的时间序列。

这绝不是一个完整的列表。大部分数据集都能被转化为更加适合分析和建模的结构化形式，虽然有时这并不是很明显。如果不行的话，也可以将数据集的特征提取为某种结构化形式。例如，一组新闻文章可以被处理为一张词频表，而这张词频表就可以用于情感分析。

大部分电子表格软件（比如Microsoft Excel，它可能是世界上使用最广泛的数据分析工具了）的用户不会对此类数据感到陌生。

为什么要使用Python进行数据分析

许许多多的人（包括我自己）都很容易爱上Python这门语言。自从1991年诞生以来，Python现在已经成为最受欢迎的动态编程语言之一，其他还有Perl、Ruby等。由于拥有大量的Web框架（比如Rails（Ruby）和Django（Python）），最近几年非常流行使用Python和Ruby进行网站建设工作。这些语言常被称作脚本（scripting）语言，因为它们可以用于编写简短而粗糙的小程序（也就是脚本）。我个人并不喜欢“脚本语言”这个术语，因为它好像在说这些语言无法用于构建严谨的软件。在众多解释型语言中，Python最大的特点是拥有一个巨大而活跃的科学计算（scientific computing）社区。进入21世纪以来，在行业应用和学术研究中采用Python进行科学计算的势头越来越猛。

在数据分析和交互、探索性计算以及数据可视化等方面，Python将不可避免地接近于其他开源和商业的领域特定编程语言/工具，如R、MATLAB、SAS、Stata等。近年来，由于Python有不断改良的库（主要是pandas），使其成为数据处理任务的一大替代方案。结合其在通用编程方面的强大实力，我们完全可以只使用Python这一种语言去构建以数据为中心的应用程序。

把Python当做粘合剂

作为一个科学计算平台，Python的成功部分源于其能够轻松地集成C、C++以及Fortran代码。大部分现代计算环境都利用了一些Fortran和C库来实现线性代数、优选、积分、快速傅里叶变换以及其他诸如此类的算法。许多企业和国家实验室也利用Python来“粘合”那些已经用了30多年的遗留软件系统。

大多数软件都是由两部分代码组成的：少量需要占用大部分执行时间的代码，以及大量不经常执行的“粘合剂代码”。粘合剂代码的执行时间通常是微不足道的。开发人员的精力几乎都是花在优化计算瓶颈上面的，有时更是直接转用更低级的语言（比如C）。

最近这几年，Cython项目（http://cython.org）已经成为Python领域中创建编译型扩展以及对接C/C++代码的一大途径。

解决“两种语言”问题

很多组织通常都会用一种类似于领域特定的计算语言（如MATLAB和R）对新的想法进行研究、原型构建和测试，然后再将这些想法移植到某个更大的生产系统中去（可能是用Java、C#或C++编写的）。人们逐渐意识到，Python不仅适用于研究和原型构建，同时也适用于构建生产系统。我相信越来越多的企业也会这样看，因为研究人员和工程技术人员使用同一种编程工具将会给企业带来非常显著的组织效益。

为什么不选Python

虽然Python非常适合构建计算密集型科学应用程序以及几乎各种各样的通用系统，但它对于不少应用场景仍然力有不逮。

由于Python是一种解释型编程语言，因此大部分Python代码都要比用编译型语言（比如Java和C++）编写的代码运行慢得多。由于程序员的时间通常都比CPU时间值钱，因此许多人也愿意在这里做一些权衡。但是，在那些要求延迟非常小的应用程序中（例如高频交易系统），为了尽最大可能地优化性能，耗费时间使用诸如C++这样更低级、更低生产率的语言进行编程也是值得的。

对于高并发、多线程的应用程序而言（尤其是拥有许多计算密集型线程的应用程序），Python并不是一种理想的编程语言。这是因为Python有一个叫做全局解释器锁（Global Interpreter Lock，GIL）的东西，这是一种防止解释器同时执行多条Python字节码指令的机制。有关“为什么会存在GIL”的技术性原因超出了本书的范围，但是就目前来看，GIL并不会在短时间内消失。虽然很多大数据处理应用程序为了能在较短的时间内完成数据集的处理工作都需要运行在计算机集群上，但是仍然有一些情况需要用单进程多线程系统来解决。

这并不是说Python不能执行真正的多线程并行代码，只不过这些代码不能在单个Python进程中执行而已。比如说，Cython项目可以集成OpenMP（一个用于并行计算的C框架）以实现并行处理循环进而大幅度提高数值算法的速度。

重要的Python库

考虑到那些还不太了解Python科学计算生态系统和库的读者，下面我先对各个库做一个简单的介绍。

NumPy

NumPy（Numerical Python的简称）是Python科学计算的基础包。本书大部分内容都基于NumPy以及构建于其上的库。它提供了以下功能（不限于此）：

·快速高效的多维数组对象ndarray。

·用于对数组执行元素级计算以及直接对数组执行数学运算的函数。

·用于读写硬盘上基于数组的数据集的工具。

·线性代数运算、傅里叶变换，以及随机数生成。

·用于将C、C++、Fortran代码集成到Python的工具。

除了为Python提供快速的数组处理能力，NumPy在数据分析方面还有另外一个主要作用，即作为在算法之间传递数据的容器。对于数值型数据，NumPy数组在存储和处理数据时要比内置的Python数据结构高效得多。此外，由低级语言（比如C和Fortran）编写的库可以直接操作NumPy数组中的数据，无需进行任何数据复制工作。

pandas

pandas提供了使我们能够快速便捷地处理结构化数据的大量数据结构和函数。你很快就会发现，它是使Python成为强大而高效的数据分析环境的重要因素之一。本书用得最多的pandas对象是DataFrame，它是一个面向列（column-oriented）的二维表结构，且含有行标和列标：

> > > frame total_bill tip sex smoker day time size1 16.99 1.01 Female No Sun Dinner 22 10.34 1.66 Male No Sun Dinner 33 21.01 3.5 Male No Sun Dinner 34 23.68 3.31 Male No Sun Dinner 25 24.59 3.61 Female No Sun Dinner 46 25.2 4.71 Male No Sun Dinner 47 8.77 2 Male No Sun Dinner 28 26.88 3.12 Male No Sun Dinner 49 15.04 1.96 Male No Sun Dinner 210 14.78 3.23 Male No Sun Dinner 2

pandas兼具NumPy高性能的数组计算功能以及电子表格和关系型数据库（如SQL）灵活的数据处理功能。它提供了复杂精细的索引功能，以便更为便捷地完成重塑、切片和切块、聚合以及选取数据子集等操作。pandas将是我在本书中使用的主要工具。

对于金融行业的用户，pandas提供了大量适用于金融数据的高性能时间序列功能和工具。事实上，我一开始就是想把pandas设计为一款适用于金融数据分析应用的工具。

对于使用R语言进行统计计算的用户，肯定不会对DataFrame这个名字感到陌生，因为它源自于R的data.frame对象。但是这两个对象并不相同。R的data.frame对象所提供的功能只是DataFrame对象所提供的功能的一个子集。虽然本书讲的是Python，但我偶尔还是会用R做对比，因为它毕竟是最流行的开源数据分析环境，而且很多读者都对它很熟悉。

pandas这个名字本身源自于panel data（面板数据，这是计量经济学中关于多维结构化数据集的一个术语）以及Python data analysis（Python数据分析）。

matplotlib

matplotlib是最流行的用于绘制数据图表的Python库。它最初由John D.Hunter（JDH）创建，目前由一个庞大的开发人员团队维护。它非常适合创建出版物上用的图表。它跟IPython（马上就会讲到）结合得很好，因而提供了一种非常好用的交互式数据绘图环境。绘制的图表也是交互式的，你可以利用绘图窗口中的工具栏放大图表中的某个区域或对整个图表进行平移浏览。

IPython

IPython是Python科学计算标准工具集的组成部分，它将其他所有的东西联系到了一起。它为交互式和探索式计算提供了一个强健而高效的环境。它是一个增强的Python shell，目的是提高编写、测试、调试Python代码的速度。它主要用于交互式数据处理和利用matplotlib对数据进行可视化处理。我在用Python编程时，经常会用到IPython，包括运行、调试和测试代码。

除标准的基于终端的IPython shell外，该项目还提供了：

·一个类似于Mathematica的HTML笔记本（通过Web浏览器连接IPython，稍后将对此进行详细介绍）。

·一个基于Qt框架的GUI控制台，其中含有绘图、多行编辑以及语法高亮显示等功能。

·用于交互式并行和分布式计算的基础架构。

我将在一章中专门讲解IPython，详细地介绍其大部分功能。强烈建议在阅读本书的过程中使用IPython。

SciPy

SciPy是一组专门解决科学计算中各种标准问题域的包的集合，主要包括下面这些包：

·scipy.integrate：数值积分例程和微分方程求解器。

·scipy.linalg：扩展了由numpy.linalg提供的线性代数例程和矩阵分解功能。

·scipy.optimize：函数优化器（最小化器）以及根查找算法。

·scipy.signal：信号处理工具。

·scipy.sparse：稀疏矩阵和稀疏线性系统求解器。

·scipy.special：SPECFUN（这是一个实现了许多常用数学函数（如伽玛函数）的Fortran库）的包装器。

·scipy.stats：标准连续和离散概率分布（如密度函数、采样器、连续分布函数等）、各种统计检验方法，以及更好的描述统计法。

·scipy.weave：利用内联C++代码加速数组计算的工具。

NumPy跟SciPy的有机结合完全可以替代MATLAB的计算功能（包括其插件工具箱）。

安装和设置

由于人们用Python所做的事情不同，所以没有一个普适的Python及其插件包的安装方案。由于许多读者的Python科学计算环境都不能完全满足本书的需要，所以接下来我将详细介绍各个操作系统上的安装方法。我建议使用下列Python安装包之一：

·Enthought Python Distribution译注1：来自Enthought（http://continumm.io/downloads）的面向科学计算的Python安装包。包括EPDFree（免费的基本版，带有NumPy、SciPy、matplotlib、Chaco以及IPython）和EPD Full（完整版，含有100多个针对各种领域的科学计算包）。EPD Full对高校免费，非高校用户需要缴纳年费。

·Python(x,y)（http://pythonxy.googlecode.com）：Windows平台上免费的Python科学计算安装包。

我将用EPDFree来说明安装过程，当然如果有需要的话，你也可以选择其他产品。编写本书时，EPD用的是Python 2.7，今后可能会有些变动。安装完毕之后，你将可以用到下面这些包：

·Python科学计算基础库：NumPy、SciPy、matplotlib以及IPython。这些都包含在EPDFree中了。

·IPython Notebook依赖项：tornado和pyzmq。这些也都包含在EPDFree中了。

·pandas（0.8.2版或更高版本）。

在阅读本书的过程中，你可能还需要安装：statsmodels、PyTables、PyQt（PySide也行）、xlrd、lxml、basemap、pymongo以及requests等（它们被用在不同的示例中）。现在暂时还不需要安装这些库，我建议你在需要的时候再安装。例如，在OS X或Linux上安装PyQt或PyTables可能会很困难。目前最重要的事情是先用EPDFree和pandas这种最小配置运行起来再说。

关于各个Python库及其安装文件和帮助信息，请访问Python Package Index（即PyPI，http://pypi.python.org）。你还可以在这里找到不少新的Python库。

注意： 为了简单起见，我将不会讨论pip译注2和virtualenv这类比较复杂的环境管理工具。网上可以找到许多介绍这些工具的优秀教程。

警告： 有些用户可能会对诸如IronPython、Jython、PyPy之类的Python实现感兴趣。为了使用本书所介绍的那些工具，（目前）就需要使用基于C的标准Python解释器（也就是CPython）。

Windows

先从http://www.enthought.com下载EPDFree的安装包，它可能是一个名字类似于epd_free-7.3-1-win-x86.msi的MSI安装包[译注3](#809468440711498-Yi_Zhu_3_You_Yu_Ruan_Jian_Ban_Ben_Geng_Xin_Jiao_Kuai__Suo_Yi_Jian_Yi_Dao_Wang_Shang_Zhao_Yi_Ge_Yi_Mo_Yi_Yang_De_An_Zhuang_Bao__Bu_Ran_You_Xie_Li_Zi_De_Jie_Guo_Ke_Neng_Hui_Gen_Shu_Shang_Jie____)。运行该安装包并接受默认的安装位置C:\Python27。如果你之前在这里安装过Python，可能需要先将其删除（可以手工删除，也可以使用控制面板中的“添加/或删除程序”功能）。

接下来，你需要验证是否已经成功将Python添加到系统路径，并且没有跟早期安装的Python版本发生冲突。首先，打开命令提示符（打开“开始”菜单，启动“命令提示符”应用程序，即cmd.exe）。输入python尝试启动Python解释器。你应该可以看到与已安装的EPDFree版本相匹配的一段消息：

C:\Users\Wes>pythonPython 2.7.3 |EPD_free 7.3-1 (32-bit)| (default, Apr 12 2012, 14:30:37) on win32Type "credits", "demo" or "enthought" for more information.>>>

如果你看到的是其他版本的EPD信息或根本什么也看不到，那就需要清理Windows环境变量。在Windows 7上，可以在程序搜索框中输入"environment variables"，然后编辑你的账户下的环境变量。在Windows XP上，需要进入“控制面板”→“系统”→“高级”→“环境变量”。在弹出窗口中找到Path变量。它需要含有下面这两个以分号隔开的目录路径：

C:\Python27;C:\Python27\Scripts

如果你之前安装了其他版本的Python，那就需要删除系统和用户Path变量中与之相关的一切路径。修改路径之后，需要重启命令提示符才能使修改生效。

能够在命令提示符中成功启动Python之后，就该安装pandas了。最简单的办法就是直接到http://pypi.python.org/pypi/pandas下载合适的二进制安装包。对于EPDFree，应该选择pandas-0.9.0.win32-py2.7.exe。将其安装好之后，接下来启动IPython来验证一下是不是万事俱备了：引入pandas，然后绘制一个简单的matplotlib图形。

C:\Users\Wes>ipython --pylabPython 2.7.3 |EPD_free 7.3-1 (32-bit)|Type "copyright", "credits" or "license" for more information.IPython 0.12.1 -- An enhanced Interactive Python.? -> Introduction and overview of IPython's features.%quickref -> Quick reference.help -> Python's own help system.object? -> Details about 'object', use 'object??' for extra details.Welcome to pylab, a matplotlib-based Python environment [backend: WXAgg]. For more information, type 'help(pylab)'.In [1]: import pandasIn [2]: plot(arange(10))

如果成功，就不会出现错误信息，而且会弹出一个绘图窗口。还可以输入下列指令译注4来检查IPython HTML notebook是否安装成功：

$ ipython notebook --pylab=inline

警告：如果你是在Windows上使用IPython notebook应用程序而且通常使用的是Internet Explorer的话，那你可能需要改用Mozilla Firefox或Google Chrome了译注5。

Windows上的EPDFree只有32位版本。如果需要使用64位版本，最简单的办法就是直接使用EPD Full译注6。如果你不想购买EPD订阅且愿意自己动手一步步安装，可以试试由加州大学欧文分校的Christoph Gohlke提供的非官方安装包（http://www.lfd.uci.edu/~gohlke/pythonlibs/），它既有32位版也有64位版，且包含本书所需的所有库。

苹果OS X

在OS X上，首先需要安装Xcode，它含有苹果的软件开发工具套件。我们所需的部分是gcc C和C++编译器。Xcode安装包可以在随计算机发布的OS X安装光盘中找到，也可以直接从苹果公司的网站上下载。

装好Xcode之后，到"Applications→Utilities"去启动终端（Terminal.app）。输入gcc并按回车键。你将会看到如下信息：

$ gcci686-apple-darwin10-gcc-4.2.1: no input files

现在就该安装EPDFree了。下载一个名为epd_free-7.3-1-macosx-i386.dmg的磁盘镜像文件。双击该.dmg文件以将其挂载到系统，然后双击其中的.mpkg文件来运行安装程序。

安装文件启动之后，会自动将EPDFree可执行文件的路径添加到你的.bash_profile文件中。该文件位于/Users/your_uname/.bash_profile：

Setting PATH for EPD_free-7.3-1PATH="/Library/Frameworks/Python.framework/Versions/Current/bin:${PATH}"export PATH

如果在后续步骤中遇到任何问题，首先应该检查一下你的.bash_profile，看看是否需要将上面那个目录添加进去。

现在就该安装pandas了。在终端中执行下面这条命令：

$ sudo easy_install pandasSearching for pandasReading http://pypi.python.org/simple/pandas/Reading http://pandas.pydata.orgReading http://pandas.sourceforge.netBest match: pandas 0.9.0Downloading http://pypi.python.org/packages/source/p/pandas/pandas-0.9.0.zipProcessing pandas-0.9.0.zipWriting /tmp/easy_install-H5mIX6/pandas-0.9.0/setup.cfgRunning pandas-0.9.0/setup.py -q bdist_egg --dist-dir /tmp/easy_install-H5mIX6/pandas-0.9.0/egg-dist-tmp-RhLG0zAdding pandas 0.9.0 to easy-install.pth fileInstalled /Library/Frameworks/Python.framework/Versions/7.3/lib/python2.7/site-packages/pandas-0.9.0-py2.7-macosx-10.5-i386.eggProcessing dependencies for pandasFinished processing dependencies for pandas

为了验证是否一切正常，我们以Pylab模式启动IPython，然后尝试加载pandas并绘制一张图片：

$ ipython --pylab22:29 ~/VirtualBox VMs/WindowsXP $ ipythonPython 2.7.3 |EPD_free 7.3-1 (32-bit)| (default, Apr 12 2012, 11:28:34)Type "copyright", "credits" or "license" for more information.IPython 0.12.1 -- An enhanced Interactive Python.? -> Introduction and overview of IPython's features.%quickref -> Quick reference.help -> Python's own help system.object? -> Details about 'object', use 'object??' for extra details.Welcome to pylab, a matplotlib-based Python environment [backend: WXAgg].For more information, type 'help(pylab)'.In [1]: import pandasIn [2]: plot(arange(10))

如果成功，将会弹出一个绘图窗口，其中画的是一条直线。

GNU/Linux

注意：有些（但不是全部）Linux产品自带的Python包版本较新，且可以通过内置的包管理工具（如apt）进行安装。我将详细讲解EPDFree的安装步骤，因为它在不同的Linux发行版之间是差不多的。

对于不同的Linux产品，具体的安装过程会有一些不同，我这里将以基于Debian的GNU/Linux系统（如Ubuntu和Mint）为例来进行讲解。除EPDFree之外，其他的安装过程跟OS X差不多。其安装包是一个只能在终端中执行的shell脚本。根据系统是32位还是64位，需要相应地安装x86版（32位）或x86_64版（64位）。然后你将会得到一个名为epd_free-7.3-1-rh5-x86_64.sh的文件。通过bash执行该脚本即可开始安装：

$ bash epd_free-7.3-1-rh5-x86_64.sh

在接受了许可协议之后，你需要选择EPDFree文件的存放位置。我建议将这些文件安装在你的home目录中，比如/home/wesm/epd（将wesm替换为你的用户名即可）。

安装完毕之后，你需要将EPDFree的bin目录添加到$PATH变量中去。如果你用的是bash shell（比如Ubuntu默认用的就是这个），则在你的.bashrc中加上下面这句路径添加指令：

export PATH=/home/wesm/epd/bin:$PATH

很明显，需要将/home/wesm/epd/替换为你所使用的安装目录。做完这些事情之后，你可以启动一个新的终端进程，也可以通过source ~/.bashrc重启你的.bashrc。

接下来还需要用到一个C编译器（比如gcc）。许多Linux产品都含有gcc，但有些则没有。在Debian系统中，可以执行下面这条指令来安装gcc：

sudo apt-get install gcc

如果在命令行中输入gcc，就可以看到：

$ gccgcc: no input files

现在可以安装pandas了：

$ easy_install pandas

如果你是以root权限来安装EPDFree的，就需要在命令中加上sudo并输入sudo或root密码。使用跟OS X相同的检测方式即可验证是否一切正常。

Python 2和Python 3

Python社区正在慢慢地从Python 2系列解释器过渡到Python 3系列。在Python 3.0问世以前，所有的Python代码都是向后兼容的。为了让Python语言更加先进，Python社区认为作出一些向后不兼容的修改是必要的。

本书是基于Python 2.7编写的，这是因为大部分Python科学计算社区还没有转向Python 3。好消息是，如果你碰巧正在使用Python 3.2，在学习本书的过程中一般也不会遇到什么麻烦。

集成开发环境（IDE）

当有人问我“你的标准开发环境是怎样的”时，我几乎总是回答“IPython外加一个文本编辑器”。我通常都在IPython中编写和调试程序，而且它可以交互式地处理数据，并通过可视化的方式验证某个数据操作的结果是否正确。诸如pandas和NumPy这样的库也可以轻松便捷地在这个shell中使用。

但是相对于文本编辑器，总有人会更喜欢IDE。因为它们提供了许多不错的“代码智能化”功能，比如自动完成以及快速获取函数和类的文档等。你可以试试下面这些：

·Eclipse +PyDev插件

·Python Tools for Visual Studio（针对Windows用户）

·PyCharm

·Spyder

·Komodo IDE

译注1

：已经更名为Enthought Canopy。EPDFree对应的是Enthought Canopy Express。相比来说EPDFree自然更好用，不过为了保证阅读本书时不遇到麻烦，建议按照本书介绍法操作。（其实就算按照书上的说明操作，一样会遇到不少麻烦，我会尽量给出说明。）

译注2

：虽然安装过程不大轻松，但还是建议后面装一下，因为它可以使你在安装那些库的时候更轻松愉快。

译注3

：由于软件版本更新较快，所以建议到网上找一个一模一样的安装包，不然有些例子的结果可能会跟书上介绍的不一样。

译注4

：如果只用过Windows，要注意前面的"$"，这是Linux或UNIX或Mac的默认命令提示符。本书应该就是用Mac测试代码的，所以这样的提示符不少，后面代码中还有很多，请读者注意。

译注5

：为什么？难道作者以为全世界人民都还在用IE6不成？译者使用IE9/IE10均无压力完成。

译注6

：还是建议使用32位版本的，最主要的原因仍然是“跟原书一致，以免抓狂”。

社区和研讨会

除搜索引擎之外，Python科学计算邮件列表也是很不错的资源，其上的问题几乎都会有人回答。可以看看下面这些：

·pydata：这是一个Google Group邮件列表，其中的问题都是Python数据分析和pandas方面的。

·pystatsmodels：针对与statsmodels和pandas相关的问题。

·numpy-discussion：针对与NumPy相关的问题。

·scipy-user：针对与SciPy和Python科学计算相关的问题。

我没有给出具体的URL，因为它们经常在变。通过搜索引擎即可轻松地找到它们。

全世界每年都会召开许多针对Python程序员的研讨会。PyCon和EuroPython分别是美国和欧洲最主要的Python研讨会。在阅读本书之后，如果你越来越深入地用Python进行数据分析，就可以在SciPy和EuroSciPy这两个面向科学计算的Python研讨会上找到许多“臭味相投的同道中人”。

使用本书

如果之前从未使用过Python，那你可能需要先看看本书最后的附录部分，那是一个讲解Python的语法、语言特性以及内置数据结构（如元组、列表和字典等）的简单教程。这部分内容可以看做本书其他内容的预备知识。

本书首先讲解的是IPython环境，然后简单地介绍了NumPy的关键特性，NumPy其他的高级功能则在本书最后一章讲解。然后我介绍了pandas。本书其余部分则介绍了综合运用pandas、NumPy和matplotlib（用于可视化）进行数据分析的相关知识。我尽量以增量的形式组织各种材料，但偶尔还是会出现一些跨章节的知识点。

各章的数据文件及相关材料存放在GitHub上译注7：

http://github.com/pydata/pydata-book

我强烈建议你下载这些数据，然后用各章所介绍的工具重写示例代码。我非常欢迎大家为本书的git库提供文稿、脚本、IPython笔记本以及其他各种有用的资源。

代码示例

本书大部分代码示例的输入形式和输出结果都会按照其在IPython shell中执行时的样子进行排版。

In [5]: codeOut[5]: output

有时，为了简洁明了，多个代码示例将会并排显示。这些代码示例应该从左到右进行阅读，且应该分别执行。

In [5]: code In [6]: code2Out[5]: output Out[6]: output2

示例数据

各章的示例数据都存放在GitHub上：http://github.com/pydata/pydata-book。下载这些数据的方法有二：使用git版本控制命令行程序；直接从网站上下载该GitHub库的zip文件。

为了让所有示例都能重现，我尽量使其包含所有必需的东西，但仍然可能会有一些错误或遗漏。如果出现这种情况的话，请给我发邮件：wesmckinn@gmail.com。

引入惯例

Python社区已经广泛接受了一些常用模块的命名惯例：

import numpy as npimport pandas as pdimport matplotlib.pyplot as plt

也就是说，当你看到np.arange时，就应该想到它引用的是NumPy中的arange函数。这样做的原因是：在Python软件开发过程中，不建议直接引入类似NumPy这种大型库的全部内容（from numpy import *）。

行话

由于你可能不太熟悉书中使用的一些有关编程和数据科学方面的常用术语，所以我在这里先给出其简单定义：

数据规整（Munge/Munging/Wrangling）译注8

指的是将非结构化和（或）散乱数据处理为结构化或整洁形式的整个过程。这几个词已经悄悄成为当今数据黑客们的行话了。Munge这个词跟Lunge押韵。

伪码（Pseudocode）

算法或过程的“代码式”描述，而这些代码本身并不是实际有效的源代码。

语法糖（Syntactic sugar）

这是一种编程语法，它并不会带来新的特性，但却能使代码更易读、更易写。

译注7

：拿到书就马上去下载，一来是防止链接不可用，二来是数据有点大，宽带较小的话……

译注8

：本来想不翻译的，但是原文中这几个到处混用，搞得我强迫症爆发，直接全翻译成“数据规整”。

致谢

如果没有那一大帮子人的帮助，我想我是写不出这本书的。

先说说O拎eilly的工作人员，我非常感谢我的编辑Meghan Blanchette和Julie Steele，他们在整个写作过程中给予了我大量的指导。Mike Loukides在本书的提案阶段也给予了很大的帮助。

许多人向我提供了大量的技术评论。具体点说，Martin Blais和Hugh Brown帮助我改进了本书的示例、简洁性以及内容组织形式。James Long、Drew Conway、Fernando Pérez、Brian Granger、Thomas Kluyver、Adam Klein、Josh Klein、Chang She以及Stéfan van der Walt都审阅了一章或几章，从多个角度给出了一些重要的反馈。

我从身边和数据社区中的好友那里得到了关于本书示例和数据集的大量灵感：Mike Dewar、Jeff Hammerbacher、James Johndrow、Kristian Lum、Adam Klein、Hilary Mason、Chang She以及Ashley Williams。

当然我还要感谢开源科学计算Python社区的许多大佬们，是他们建立了我开发工作的基础，在我编写本书时也给予了不少的鼓励：IPython核心团队（Fernando Pérez、Brian Granger、Min Ragan-Kelly、Thomas Kluyver等）、John Hunter、Skipper Seabold、Travis Oliphant、Peter Wang、Eric Jones、Robert Kern、Josef Perktold、Francesc Alted、Chris Fonnesbeck，还有很多人就不一一列举了。另外还有许多人在整个过程中也给予了大量的支持、建议和鼓励：Drew Conway、Sean Taylor、Giuseppe Paleologo、Jared Lander、David Epstein、John Krowas、Joshua Bloom、Den Pilsworth、John Myles-White，还有许多我都已经不记得了。

还要感谢我整个成长岁月中的一些人。首先，我原来在AQR公司的同事们，他们在我从事pandas项目时给予了不少的支持：Alex Reyfman、Michael Wong、Tim Sargen、Oktay Kurbanov、Matthew Tschantz、Roni Israelov、Michael Katz、Chris Uga、Prasad Ramanan、Ted Square，以及Hoon Kim。然后是我的导师Haynes Miller（麻省理工学院）和Mike West（杜克大学）。

私人方面，Casey Dinkin在我写书期间给予了大量的关心和照顾，并忍受了我一切的情绪波动，因为我在过了预定时间之后才东拼西凑出了最终的手稿。然后是我的父母Bill和Kim，从我很小时他们就教育我要有理想，而且绝不退而求其次。

第2章　引言

本书将要向你介绍的是用于高效处理数据的Python工具。虽然读者各自工作的最终目的千差万别，但基本都需要完成以下几个大类的任务：

与外界进行交互

读写各种各样的文件格式和数据库。

准备

对数据进行清理、修整、整合、规范化、重塑、切片切块、变形等处理以便进行分析。

转换

对数据集做一些数学和统计运算以产生新的数据集。比如说，根据分组变量对一个大表进行聚合。

建模和计算

将数据跟统计模型、机器学习算法或其他计算工具联系起来。

展示

创建交互式的或静态的图片或文字摘要。

我将在本章中给出一些范例数据集，并讲解我们能对其做些什么。这些例子仅仅是为了提起你的兴趣，因此只会在一个比较高的层次进行讲解。即使你从来没用过这些东西也没关系，本书后续的章节将会对此进行非常详细的讲解。在这些代码示例中，你可以看到诸如In [15]:之类的输入输出提示符，它们来自IPython shell。

来自bit.ly的1.usa.gov数据

2011年，URL缩短服务bit.ly跟美国政府网站usa.gov合作，提供了一份从生成.gov或.mil短链接的用户那里收集来的匿名数据译注1。直到编写本书时为止，除实时数据译注2之外，还可以下载文本文件形式的每小时快照注1。

以每小时快照为例，文件中各行的格式为JSON（即JavaScript Object Notation，这是一种常用的Web数据格式）。例如，如果我们只读取某个文件中的第一行，那么你所看到的结果应该是下面这样：

In [15]: path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'In [16]: open(path).readline()Out[16]: '{ "a": "Mozilla\/5.0 (Windows NT 6.1; WOW64) AppleWebKit\/535.11 (KHTML, like Gecko) Chrome\/17.0.963.78 Safari\/535.11", "c": "US", "nk": 1, "tz": "America\/New_York", "gr": "MA", "g": "A6qOVH", "h": "wfLQtf", "l": "orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r": "http:\/\/www.facebook.com\/l\/7AQEFzjSi\/1.usa.gov\/wfLQtf", "u": "http:\/\/www.ncbi.nlm.nih.gov\/pubmed\/22415991", "t": 1331923247, "hc":1331822918, "cy": "Danvers", "ll": [42.576698, -70.954903] }\n'

Python有许多内置或第三方模块可以将JSON字符串转换成Python字典对象。这里，我将使用json模块及其loads函数逐行加载已经下载好的数据文件：

import jsonpath = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'records = [json.loads(line) for line in open(path)]

你可能之前没用过Python，解释一下上面最后那行表达式，它叫做列表推导式（list comprehension），这是一种在一组字符串（或一组别的对象）上执行一条相同操作（如json.loads）的简洁方式。在一个打开的文件句柄上进行迭代即可获得一个由行组成的序列。现在，records对象就成为一组Python字典了：

In [18]: records[0]Out[18]:{u'a': u'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko) Chrome/17.0.963.78 Safari/535.11', u'al': u'en-US,en;q=0.8',u'c': u'US',u'cy': u'Danvers',u'g': u'A6qOVH',u'gr': u'MA',u'h': u'wfLQtf',u'hc': 1331822918,u'hh': u'1.usa.gov',u'l': u'orofrog',u'll': [42.576698, -70.954903],u'nk': 1,u'r': u'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',u't': 1331923247,u'tz': u'America/New_York',u'u': u'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}

注意，Python的索引是从0开始的，不像其他某些语言是从1开始的（如R）。现在，只要以字符串的形式给出想要访问的键就可以得到当前记录中相应的值了：

In [19]: records[0]['tz']Out[19]: u'America/New_York'

单引号前面的u表示unicode（一种标准的字符串编码格式）。注意，IPython在这里给出的是时区的字符串对象形式，而不是其打印形式：

In [20]: print records[0]['tz']America/New_York

用纯Python代码对时区进行计数

假设我们想要知道该数据集中最常出现的是哪个时区（即tz字段），得到答案的办法有很多。首先，我们用列表推导式取出一组时区：

In [25]: time_zones = [rec['tz'] for rec in records]---KeyError Traceback (most recent call last)/home/wesm/book_scripts/whetting/<ipython> in <module>()----> 1 time_zones = [rec['tz'] for rec in records]KeyError: 'tz'

晕！原来并不是所有记录都有时区字段。这个好办，只需在列表推导式末尾加上一个if 'tz'in rec判断即可：

In [26]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]In [27]: time_zones[:10]Out[27]: [u'America/New_York',u'America/Denver',u'America/New_York',u'America/Sao_Paulo',u'America/New_York',u'America/New_York',u'Europe/Warsaw',u'',u'',u'']

只看前10个时区，我们发现有些是未知的（即空的）。虽然可以将它们过滤掉，但现在暂时先留着。接下来，为了对时区进行计数，这里介绍两个办法：一个较难（只使用标准Python库），另一个较简单（使用pandas）。计数的办法之一是在遍历时区的过程中将计数值保存在字典中：

def get_counts(sequence): counts = {} for x in sequence: if x in counts: counts[x] += 1 else: counts[x] = 1 return counts

如果非常了解Python标准库，那么你可能会将代码写得更简洁一些：

from collections import defaultdictdef get_counts2(sequence): counts = defaultdict(int) # 所有的值均会被初始化为0 for x in sequence: counts[x] += 1 return counts

我将代码写到函数中是为了获得更高的可重用性。要用它对时区进行处理，只需将time_zones传入即可：

In [31]: counts = get_counts(time_zones)In [32]: counts['America/New_York']Out[32]: 1251In [33]: len(time_zones)Out[33]: 3440

如果想要得到前10位的时区及其计数值，我们需要用到一些有关字典的处理技巧：

def top_counts(count_dict, n=10): value_key_pairs = [(count, tz) for tz, count in count_dict.items()] value_key_pairs.sort() return value_key_pairs[-n:]

现在我们就可以：

In [35]: top_counts(counts)Out[35]:[(33, u'America/Sao_Paulo'), (35, u'Europe/Madrid'), (36, u'Pacific/Honolulu'), (37, u'Asia/Tokyo'), (74, u'Europe/London'), (191, u'America/Denver'), (382, u'America/Los_Angeles'), (400, u'America/Chicago'), (521, u''), (1251, u'America/New_York')]

你可以在Python标准库中找到collections.Counter类，它能使这个任务变得更简单：

In [49]: from collections import CounterIn [50]: counts = Counter(time_zones)In [51]: counts.most_common(10)Out[51]:[(u'America/New_York', 1251), (u'', 521), (u'America/Chicago', 400), (u'America/Los_Angeles', 382), (u'America/Denver', 191), (u'Europe/London', 74), (u'Asia/Tokyo', 37), (u'Pacific/Honolulu', 36), (u'Europe/Madrid', 35), (u'America/Sao_Paulo', 33)]

用pandas对时区进行计数

DataFrame是pandas中最重要的数据结构，它用于将数据表示为一个表格。从一组原始记录中创建DataFrame是很简单的：

In [289]: from pandas import DataFrame, SeriesIn [290]: import pandas as pd; import numpy as npIn [291]: frame = DataFrame(records)In [292]: frameOut[292]:Int64Index: 3560 entries, 0 to 3559Data columns:*heartbeat* 120 non-null valuesa 3440 non-null valuesal 3094 non-null

valuesc 2919 non-null valuescy 2919 non-null valuesg 3440 non-null valuesgr 2919 non-null valuesh 3440 non-null valueshc 3440 non-null valueshh 3440 non-null valueskw 93 non-null valuesl 3440 non-null valuesll 2919 non-null valuesnk 3440 non-null valuesr 3440 non-null valuest 3440 non-null valuestz 3440 non-null valuesu 3440 non-null valuesdtypes: float64(4), object(14)In [293]: frame['tz'][:10]Out[293]:0 America/New_York1 America/Denver2 America/New_York3 America/Sao_Paulo4 America/New_York5 America/New_York6 Europe/Warsaw789Name: tz

这里frame的输出形式是摘要视图（summary view），主要用于较大的DataFrame对象。frame['tz']所返回的Series对象有一个value_counts方法，该方法可以让我们得到所需的信息：

In [294]: tz_counts = frame['tz'].value_counts()In [295]: tz_counts[:10]Out[295]:America/New_York 1251 521America/Chicago 400America/Los_Angeles 382America/Denver 191Europe/London 74Asia/Tokyo 37Pacific/Honolulu 36Europe/Madrid 35America/Sao_Paulo 33

然后，我们想利用绘图库（即matplotlib）为这段数据生成一张图片。为此，我们先给记录中未知或缺失的时区填上一个替代值。fillna函数可以替换缺失值（NA），而未知值（空字符串）则可以通过布尔型数组索引加以替换：

In [296]: clean_tz = frame['tz'].fillna('Missing')In [297]: clean_tz[clean_tz == ''] = 'Unknown'In [298]: tz_counts = clean_tz.value_counts()In [299]: tz_counts[:10]Out[299]:America/New_York 1251Unknown 521America/Chicago 400America/Los_Angeles 382America/Denver 191Missing 120Europe/London 74Asia/Tokyo 37Pacific/Honolulu 36Europe/Madrid 35

利用counts译注3对象的plot方法即可得到一张水平条形图译注4：

In [301]: tz_counts[:10].plot(kind='barh', rot=0)

最终结果如图2-1所示。我们还可以对这种数据进行很多处理。比如说，a字段含有执行URL短缩操作的浏览器、设备、应用程序的相关信息：

In [302]: frame['a'][1]Out[302]: u'GoogleMaps/RochesterNY'In [303]: frame['a'][50]Out[303]: u'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2'In [304]: frame['a'][51]Out[304]: u'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P925/V10e Build/FRG83G) AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1'

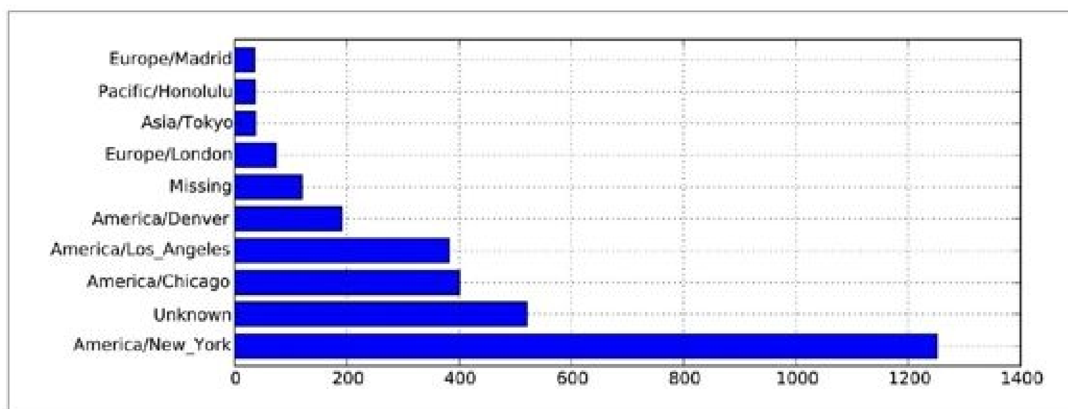


图2-1：1.usa.gov示例数据中最常出现的时区

将这些"agent"字符串译注5中的所有信息都解析出来是一件挺郁闷的工作。不过只要你掌握了Python内置的字符串函数和正则表达式，事情就好办了。比如说，我们可以将这种字符串的第一节（与浏览器大致对应）分离出来并得到另外一份用户行为摘要：

In [305]: results = Series([x.split()[0] for x in frame.a.dropna()])In [306]: results[:5]Out[306]:0 Mozilla/5.01 GoogleMaps/RochesterNY2 Mozilla/4.03 Mozilla/5.04 Mozilla/5.0In [307]: results.value_counts()[:8]Out[307]:Mozilla/5.0 2594Mozilla/4.0 601GoogleMaps/RochesterNY 121Opera/9.80 34TEST_INTERNET_AGENT 24GoogleProducer 21Mozilla/6.0 5BlackBerry8520/5.0.0.681 4

现在，假设你想按Windows和非Windows用户对时区统计信息进行分解。为了简单起见，我们假定只要agent字符串中含有"Windows"就认为该用户为Windows用户。由于有的agent缺失，所以首先将它们从数据中移除：

In [308]: cframe = frame[frame.a.notnull()]

其次根据a值计算出各行是否是Windows：

In [309]: operating_system = np.where(cframe['a'].str.contains('Windows'), ...: 'Windows', 'Not Windows')In [310]: operating_system[:5]Out[310]:0 Windows1 Not Windows2 Windows3 Not Windows4 WindowsName: a

接下来就可以根据时区和新得到的操作系统列表对数据进行分组了：

In [311]: by_tz_os = cframe.groupby(['tz', operating_system])

然后通过size对分组结果进行计数（类似于上面的value_counts函数），并利用unstack对计数结果进行重塑：

In [312]: agg_counts = by_tz_os.size().unstack().fillna(0)In [313]: agg_counts[:10]Out[313]:a Not Windows Windowstz 245 276Africa/Cairo 0 3Africa/Casablanca 0 1Africa/Ceuta 0 2Africa/Johannesburg 0 1Africa/Lusaka 0 1America/Anchorage 4 1America/Argentina/Buenos_Aires 1 0America/Argentina/Cordoba 0 1America/Argentina/Mendoza 0 1

最后，我们来选取最常出现的时区。为了达到这个目的，我根据agg_counts中的行数构造了一个间接索引数组：

用于按升序排列In [314]: indexer = agg_counts.sum(1).argsort()In [315]: indexer[:10]Out[315]:tz 24Africa/Cairo 20Africa/Casablanca 21Africa/Ceuta 92Africa/Johannesburg 87Africa/Lusaka 53America/Anchorage 54America/Argentina/Buenos_Aires 57America/Argentina/Cordoba 26America/Argentina/Mendoza 55

然后我通过take按照这个顺序截取了最后10行：

In [316]: count_subset = agg_counts.take(indexer)[-10:]In [317]: count_subsetOut[317]:a Not Windows WindowstzAmerica/Sao_Paulo 13 20Europe/Madrid 16 19Pacific/Honolulu 0 36Asia/Tokyo 2 35Europe/London 43 31America/Denver 132 59America/Los_Angeles 130 252America/Chicago 115 285 245 276America/New_York 339 912

这里也可以生成一张条形图。我将使用stacked=True来生成一张堆积条形图（如图2-2所示）：

In [319]: count_subset.plot(kind='barh', stacked=True)

由于在这张图中不太容易看清楚较小分组中Windows用户的相对比例，因此我们可以将各行规范化为“总计为1”并重新绘图（如图2-3所示）：

In [321]: normed_subset = count_subset.div(count_subset.sum(1), axis=0)In [322]: normed_subset.plot(kind='barh', stacked=True)

这里所用到的所有方法都会在本书后续的章节中详细讲解。

译注1

：由于可以通过短链接伪造.gov后缀的URL，导致用户访问恶意域名，所以美国政府开始着手处理这种事情了。

译注2

：以Feed形式提供。

注1

：网址：http://www.usa.gov/About/developer-resources/1usagov.shtml。

译注3

：应该是tz_counts 。

译注4

：注意，一定要以pylab模式打开，否则这条代码没效果。包括很多缩写，pylab都直接弄好了，如果不是用这种模式打开，后面很多代码一样会遇到问题，虽然不是什么大毛病，但毕竟麻烦。后面如果遇到这没定义那找不到的情况，就请注意是不是因为这个。

译注5

：即浏览器的USER_AGENT信息。

MovieLens 1M数据集

GroupLens Research（http://www.grouplens.org/node/73）采集了一组从20世纪90年末到21世纪初由MovieLens用户提供的电影评分数据。这些数据中包括电影评分、电影元数据（风格类型和年代）以及关于用户的人口统计学数据（年龄、邮编、性别和职业等）。基于机器学习算法的推荐系统一般都会对此类数据感兴趣。虽然我不会在本书中详细介绍机器学习技术，但我会告诉你如何对这种数据进行切片切块以满足实际需求。

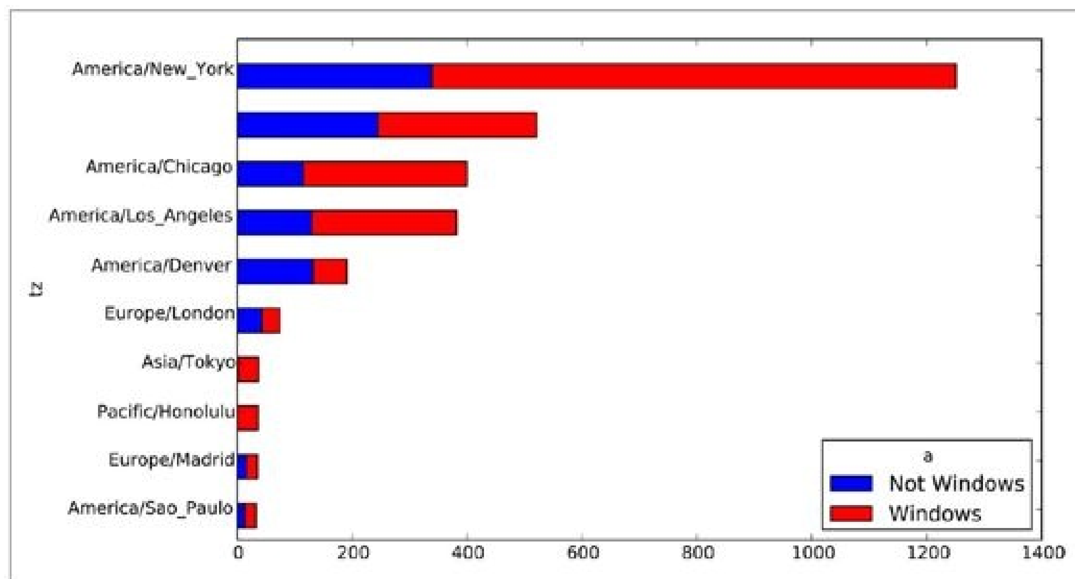


图2-2：按Windows和非Windows用户统计的最常出现的时区

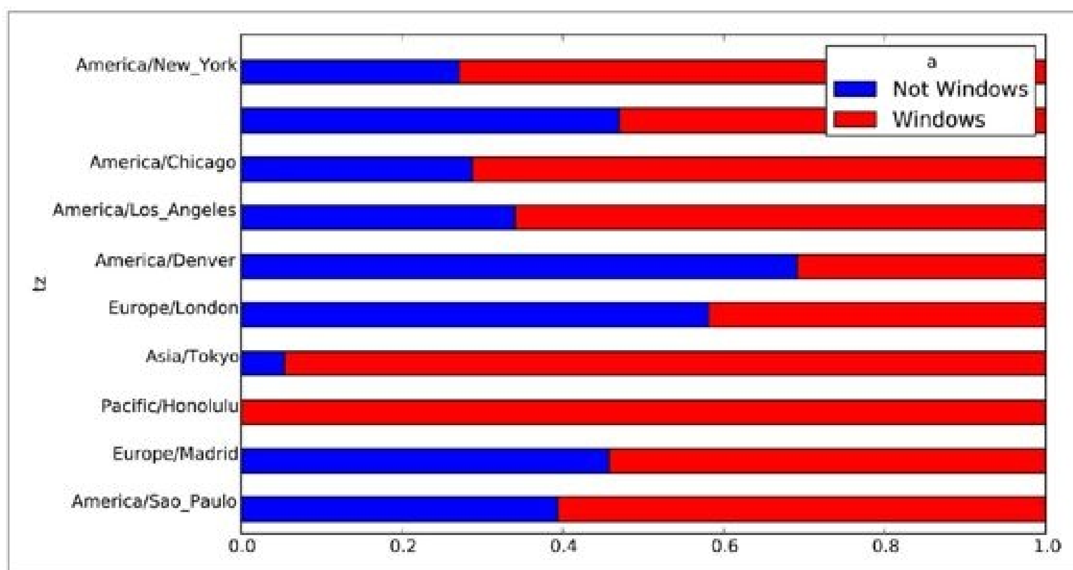


图2-3：按Windows和非Windows用户比例统计的最常出现的时区

MovieLens 1M数据集含有来自6000名用户对4000部电影的100万条评分数据。它分为三个表：评分、用户信息和电影信息。将该数据从zip文件中解压出来之后，可以通过pandas.read_table将各个表分别读到一个pandas DataFrame对象中：

import pandas as pdunames = ['user_id', 'gender', 'age', 'occupation', 'zip']users = pd.read_table('ml-1m/users.dat', sep='::', header=None, names=unames)rnames = ['user_id', 'movie_id', 'rating', 'timestamp']ratings = pd.read_table('ml-1m/ratings.dat', sep='::', header=None, names=rnames)mnames = ['movie_id', 'title', 'genres']movies = pd.read_table('ml-1m/movies.dat', sep='::', header=None, names=mnames)

利用Python的切片语法，通过查看每个DataFrame的前几行即可验证数据加载工作是否一切顺利：

In [334]: users[:5]Out[334]: user_id gender age occupation zip0 1 F 1 10 480671 2 M 56 16 700722 3 M 25 15 551173 4 M 45 7 024604 5 M 25 20 55455In [335]: ratings[:5]Out[335]: user_id movie_id rating timestamp0 1 1193 5 9783007601 1 661 3 9783021092 1 914 3 9783019683 1 3408 4 9783002754 1 2355 5 978824291In [336]: movies[:5]Out[336]: movie_id title genres0 1 Toy Story (1995) Animation|Children's|Comedy1 2 Jumanji (1995) Adventure|Children's|Fantasy2 3 Grumpier Old Men (1995) Comedy|Romance3 4 Waiting to Exhale (1995) Comedy|Drama4 5 Father of the Bride Part II (1995) ComedyIn [337]: ratingsOut[337]:Int64Index: 1000209 entries, 0 to 1000208Data columns:user_id 1000209 non-null valuesmovie_id 1000209 non-null valuesrating 1000209 non-null valuestimestamp 1000209 non-null valuesdtypes: int64(4)

注意，其中的年龄和职业是以编码形式给出的，它们的具体含义请参考该数据集的README文件。分析散布在三个表中的数据可不是一件轻松的事情。假设我们想要根据性别和年龄计算某部电影的平均得分，如果将所有数据都合并到一个表中的话问题就简单多了。我们先用

pandas的merge函数将ratings跟users合并到一起，然后再将movies也合并进去。pandas会根据列名的重叠情况推断出哪些列是合并（或连接）键：

In [338]: data = pd.merge(pd.merge(ratings, users), movies)In [339]: dataOut[339]:Int64Index: 1000209 entries, 0 to 1000208Data columns:user_id 1000209 non-null valuesmovie_id 1000209 non-null valuesrating 1000209 non-null valuestimestamp 1000209 non-null valuesgender 1000209 non-null valuesage 1000209 non-null valuesoccupation 1000209 non-null valueszip 1000209 non-null valuestitle 1000209 non-null valuesgenres 1000209 non-null valuesdtypes: int64(6), object(4)In [340]: data.ix[0]Out[340]:user_id 1movie_id 1rating 5timestamp 978824268gender Fage 1occupation 10zip 48067title Toy Story (1995)genres Animation|Children's|ComedyName: 0

现在，只要稍微熟悉一下pandas，就能轻松地根据任意个用户或电影属性对评分数据进行聚合操作了。为了按性别计算每部电影的平均得分，我们可以使用pivot_table方法：

In [341]: mean_ratings = data.pivot_table('rating', rows='title', ...: cols='gender', aggfunc='mean')In [342]: mean_ratings[:5]Out[342]:gender F Mtitle$1,000,000 Duck (1971) 3.375000 2.761905'Night Mother (1986) 3.388889 3.352941'Til There Was You (1997) 2.675676 2.733333'burbs, The (1989) 2.793478 2.962085...And Justice for All (1979) 3.828571 3.689024

该操作产生了另一个DataFrame，其内容为电影平均得分，行标为电影名称，列标为性别。现在，我打算过滤掉评分数据不够250条的电影（随便选的一个数字）。为了达到这个目的，我先对title进行分组，然后利用size()得到一个含有各电影分组大小的Series对象：

In [343]: ratings_by_title = data.groupby('title').size()In [344]: ratings_by_title[:10]Out[344]:title$1,000,000 Duck (1971) 37'Night Mother (1986) 70'Til There Was You (1997) 52'burbs, The (1989) 303...And Justice for All (1979) 1991-900 (1994) 210 Things I Hate About You (1999) 700101 Dalmatians (1961) 565101 Dalmatians (1996) 36412 Angry Men (1957) 616In [345]: active_titles = ratings_by_title.index[ratings_by_title >= 250]In [346]: active_titlesOut[346]:Index(['burbs, The (1989), 10 Things I Hate About You (1999), 101 Dalmatians (1961), ..., Young Sherlock Holmes (1985), Zero Effect (1998), eXistenZ (1999)], dtype=object)

该索引中含有评分数据大于250条的电影名称，然后我们就可以据此从前面的mean_ratings中选取所需的行了：

In [347]: mean_ratings = mean_ratings.ix[active_titles]In [348]: mean_ratingsOut[348]:Index: 1216 entries, 'burbs, The (1989) to eXistenZ (1999)Data columns:F 1216 non-null valuesM 1216 non-null valuesdtypes: float64(2)

为了了解女性观众最喜欢的电影，我们可以对F列降序排列：

In [350]: top_female_ratings = mean_ratings.sort_index(by='F', ascending=False)In [351]: top_female_ratings[:10]Out[351]:gender F MtitleClose Shave, A (1995) 4.644444 4.473795Wrong Trousers, The (1993) 4.588235 4.478261Sunset Blvd. (a.k.a. Sunset Boulevard) (1950) 4.572650 4.464589Wallace & Gromit: The Best of Aardman Animation (1996) 4.563107 4.385075Schindler's List (1993) 4.562602 4.491415Shawshank Redemption, The (1994) 4.539075 4.560625Grand Day Out, A (1992) 4.537879 4.293255To Kill a Mockingbird (1962) 4.536667 4.372611Creature Comforts (1990) 4.513889 4.272277Usual Suspects, The (1995) 4.513317 4.518248

计算评分分歧

假设我们想要找出男性和女性观众分歧最大的电影。一个办法是给mean_ratings加上一个用于存放平均得分之差的列，并对其进行排序：

In [352]: mean_ratings['diff'] = mean_ratings['M'] - mean_ratings['F']

按"diff"排序即可得到分歧最大且女性观众更喜欢的电影：

In [353]: sorted_by_diff = mean_ratings.sort_index(by='diff')In [354]: sorted_by_diff[:15]Out[354]:gender F M difftitleDirty Dancing (1987) 3.790378 2.959596 -0.830782Jumpin' Jack Flash (1986) 3.254717 2.578358 -0.676359Grease (1978) 3.975265 3.367041 -0.608224Little Women (1994) 3.870588 3.321739 -0.548849Steel Magnolias (1989) 3.901734 3.365957 -0.535777Anastasia (1997) 3.800000 3.281609 -0.518391Rocky Horror Picture Show, The (1975) 3.673016 3.160131 -0.512885Color Purple, The (1985) 4.158192 3.659341 -0.498851Age of Innocence, The (1993) 3.827068 3.339506 -0.487561Free Willy (1993) 2.921348 2.438776 -0.482573French Kiss (1995) 3.535714 3.056962 -0.478752Little Shop of Horrors, The (1960) 3.650000 3.179688 -0.470312Guys and Dolls (1955) 4.051724 3.583333 -0.468391Mary Poppins (1964) 4.197740 3.730594 -0.467147Patch Adams (1998) 3.473282 3.008746 -0.464536

对排序结果反序并取出前15行，得到的则是男性观众更喜欢的电影：

对行反序，并取出前15行In [355]: sorted_by_diff[::-1][:15]Out[355]:gender F M difftitleGood, The Bad and The Ugly, The (1966) 3.494949 4.221300 0.726351Kentucky Fried Movie, The (1977) 2.878788 3.555147 0.676359Dumb & Dumber (1994) 2.697987 3.336595

0.638608Longest Day, The (1962) 3.411765 4.031447 0.619682Cable Guy, The (1996) 2.250000 2.863787 0.613787Evil Dead II (Dead By Dawn) (1987) 3.297297 3.909283 0.611985Hidden, The (1987) 3.137931 3.745098 0.607167Rocky III (1982) 2.361702 2.943503 0.581801Caddyshack (1980) 3.396135 3.969737 0.573602For a Few Dollars More (1965) 3.409091 3.953795 0.544704Porky's (1981) 2.296875 2.836364 0.539489Animal House (1978) 3.628906 4.167192 0.538286Exorcist, The (1973) 3.537634 4.067239 0.529605Fright Night (1985) 2.973684 3.500000 0.526316Barb Wire (1996) 1.585366 2.100386 0.515020

如果只是想要找出分歧最大的电影（不考虑性别因素），则可以计算得分数据的方差或标准差：

根据电影名称分组的得分数据的标准差In [356]: rating_std_by_title = data.groupby('title')['rating'].std()# 根据active_titles进行过滤In [357]: rating_std_by_title = rating_std_by_title.ix[active_titles]# 根据值对Series进行降序排列In [358]: rating_std_by_title.order(ascending=False

)[:10]Out[358]:titleDumb & Dumber (1994) 1.321333Blair Witch Project, The (1999) 1.316368Natural Born Killers (1994) 1.307198Tank Girl (1995) 1.277695Rocky Horror Picture Show, The (1975) 1.260177Eyes Wide Shut (1999) 1.259624Evita (1996) 1.253631Billy Madison (1995) 1.249970Fear and Loathing in Las Vegas (1998) 1.246408Bicentennial Man (1999) 1.245533Name: rating

可能你已经注意到了，电影分类是以竖线（|）分隔的字符串形式给出的。如果想对电影分类进行分析的话，就需要先将其转换成更有用的形式才行。我将在本书后续章节中讲到这种转换处理，到时还会再用到这个数据。

1880-2010年间全美婴儿姓名

美国社会保障总署（SSA）提供了一份从1880年到2010年的婴儿名字频率数据。Hadley Wickham（许多流行R包的作者）经常用这份数据来演示R的数据处理功能。

In [4]: names.head(10)Out[4]: name sex births year0 Mary F 7065 18801 Anna F 2604 18802 Emma F 2003 18803 Elizabeth F 1939 18804 Minnie F 1746 18805 Margaret F 1578 18806 Ida F 1472 18807 Alice F 1414 18808 Bertha F 1320 18809 Sarah F 1288 1880

你可以用这个数据集做很多事，例如：

·计算指定名字（可以是你自己的，也可以是别人的）的年度比例。

·计算某个名字的相对排名。

·计算各年度最流行的名字，以及增长或减少最快的名字。

·分析名字趋势：元音、辅音、长度、总体多样性、拼写变化、首尾字母等。

·分析外源性趋势：圣经中的名字、名人、人口结构变化等。

利用前面介绍过的那些工具，这些分析工作都能很轻松地完成，因此我会尽量多讲一些。我建议你下载这些数据并亲自试一试。如果你在这些数据中找到了某个有趣的模式，我将非常乐意听上一听。

到编写本书时为止，美国社会保障总署将该数据库按年度制成了多个数据文件，其中给出了每个性别/名字组合的出生总数。这些文件的原始档案可以在这里获取：译注6

http://www.ssa.gov/oact/babynames/limits.html?

如果你在阅读本书的时候这个页面已经不见了，也可以用搜索引擎找找。下载"National data"文件names.zip，解压后的目录中含有一组文件（如yob1880.txt）。我用UNIX的head命令查看了其中一个文件的前10行（在Windows上，你可以用more命令，或直接在文本编辑器中打开）：

In [367]: !head -n 10 names/yob1880.txtMary,F,7065Anna,F,2604Emma,F,2003Elizabeth,F,1939Minnie,F,1746Margaret,F,1578Ida,F,1472Alice,F,1414Bertha,F,1320Sarah,F,1288

由于这是一个非常标准的以逗号隔开的格式，所以可以用pandas.read_csv将其加载到DataFrame中：

In [368]: import pandas as pdIn [369]: names1880 = pd.read_csv('names/yob1880.txt', names=['name', 'sex', 'births'])In [370]: names1880Out[370]:Int64Index: 2000 entries, 0 to 1999Data columns:name 2000 non-null valuessex 2000 non-null valuesbirths 2000 non-null valuesdtypes: int64(1), object(2)

这些文件中仅含有当年出现超过5次的名字。为了简单起见，我们可以用births列的sex分组小计表示该年度的births总计：

In [371]: names1880.groupby('sex').births.sum()Out[371]:sexF 90993M 110493Name: births

由于该数据集按年度被分隔成了多个文件，所以第一件事情就是要将所有数据都组装到一个DataFrame里面，并加上一个year字段。使用pandas.concat即可达到这个目的：

2010是目前最后一个有效统计年度years = range(1880, 2011)pieces = []columns = ['name', 'sex', 'births']for year in years: path = 'names/yob%d.txt' % year frame = pd.read_csv(path, names=columns) frame['year'] = year pieces.append(frame)# 将所有数据整合到单个DataFrame中names = pd.concat(pieces, ignore_index=True)

这里需要注意几件事情。第一，concat默认是按行将多个DataFrame组合到一起的；第二，必须指定ignore_index=True，因为我们不希望保留read_csv所返回的原始行号。现在我们得到了一个非常大的DataFrame，它含有全部的名字数据。

现在names这个DataFrame对象看上去应该是这个样子：

In [373]: namesOut[373]:Int64Index: 1690784 entries, 0 to 1690783Data columns:name 1690784 non-null valuessex 1690784 non-null valuesbirths 1690784 non-null valuesyear 1690784 non-null valuesdtypes: int64(2), object(2)

有了这些数据之后，我们就可以利用groupby或pivot_table在year和sex级别上对其进行聚合了，如图2-4所示：

In [374]: total_births = names.pivot_table('births', rows='year', ...: cols='sex', aggfunc=sum)In [375]: total_births.tail()Out[375]:sex F Myear2006 1896468 20502342007 1916888 20692422008 1883645 20323102009 1827643 19733592010 1759010 1898382In [376]: total_births.plot(title='Total births by sex and year')

下面我们来插入一个prop列，用于存放指定名字的婴儿数相对于总出生数的比例。prop值为0.02表示每100名婴儿中有2名取了当前这个名字。因此，我们先按year和sex分组，然后再将新列加到各个分组上：

def add_prop(group): # 整数除法会向下圆整 births = group.births.astype(float) group['prop'] = births / births.sum()   return groupnames = names.groupby(['year', 'sex']).apply(add_prop)

注意：由于births是整数，所以我们在计算分式时必须将分子或分母转换成浮点数（除非你正在使用Python 3！）。

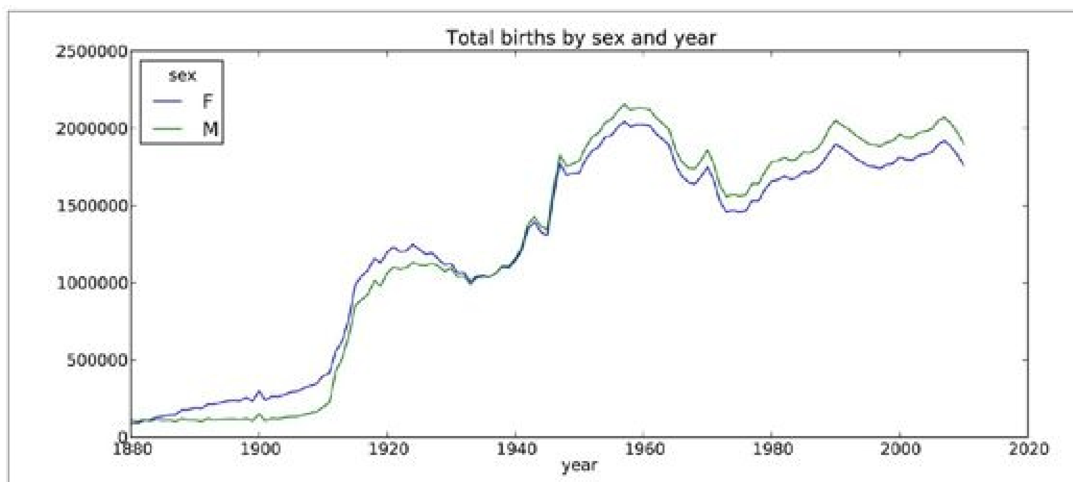


图2-4：按性别和年度统计的总出生数

现在，完整的数据集就有了下面这些列：

In [378]: namesOut[378]:Int64Index: 1690784 entries, 0 to 1690783Data columns:name 1690784 non-null valuessex 1690784 non-null valuesbirths 1690784 non-null valuesyear 1690784 non-null valuesprop 1690784 non-null valuesdtypes: float64(1), int64(2), object(2)

在执行这样的分组处理时，一般都应该做一些有效性检查，比如验证所有分组的prop的总和是否为1。由于这是一个浮点型数据，所以我们应该用np.allclose来检查这个分组总计值是否足够近似于（可能不会精确等于）1：

In [379]: np.allclose(names.groupby(['year', 'sex']).prop.sum(), 1)Out[379]: True

这样就算完活了。为了便于实现更进一步的分析，我需要取出该数据的一个子集：每对sex/year组合的前1000个名字。这又是一个分组操作：

def get_top1000(group): return group.sort_index(by='births', ascending=False)[:1000]grouped = names.groupby(['year', 'sex'])top1000 = grouped.apply(get_top1000)

如果你喜欢DIY的话，也可以这样：

pieces = []for year, group in names.groupby(['year', 'sex']): pieces.append(group.sort_index(by='births', scending=False)[:1000])top1000 = pd.concat(pieces, ignore_index=True)

现在的结果数据集就小多了：

In [382]: top1000Out[382]:Int64Index: 261877 entries, 0 to 261876Data columns:name 261877 non-null valuessex 261877 non-null valuesbirths 261877 non-null valuesyear 261877 non-null valuesprop 261877 non-null valuesdtypes: float64(1), int64(2), object(2)

接下来的数据分析工作就针对这个top1000数据集了。

分析命名趋势

有了完整的数据集和刚才生成的top1000数据集，我们就可以开始分析各种命名趋势了。首先将前1000个名字分为男女两个部分：

In [383]: boys = top1000[top1000.sex == 'M']In [384]: girls = top1000[top1000.sex == 'F']

这是两个简单的时间序列，只需稍作整理即可绘制出相应的图表（比如每年叫做John和Mary的婴儿数）。我们先生成一张按year和name统计的总出生数透视表：

In [385]: total_births = top1000.pivot_table('births', rows='year', cols='name', ...: aggfunc=sum)

现在，我们用DataFrame的plot方法绘制几个名字的曲线图：

In [386]: total_birthsOut[386]:Int64Index: 131 entries, 1880 to 2010Columns: 6865 entries, Aaden to Zuridtypes: float64(6865)In [387]: subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]In [388]: subset.plot(subplots=True, figsize=(12, 10), grid=False, ...: title="Number

of births per year")

最终结果如图2-5所示。从图中可以看出，这几个名字在美国人民的心目中已经风光不再了。但事实并非如此简单，我们在下一节中就能知道是怎么一回事了。

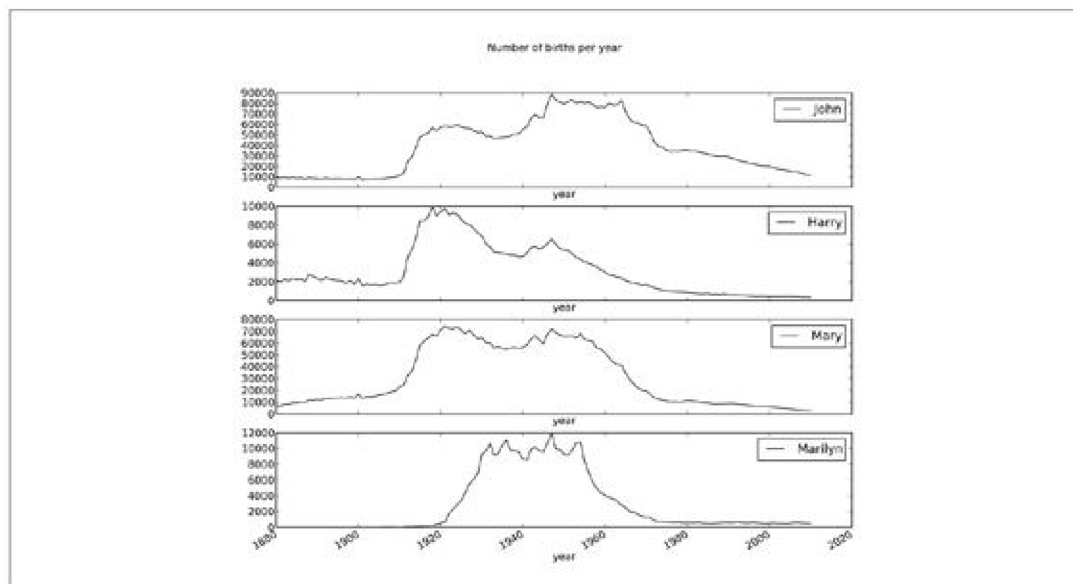


图2-5：几个男孩和女孩名字随时间变化的使用数量

评估命名多样性的增长

图2-5所反映的降低情况可能意味着父母愿意给小孩起常见的名字越来越少。这个假设可以从数据中得到验证。一个办法是计算最流行的1000个名字所占的比例，我按year和sex进行聚合并绘图：

In [390]: table = top1000.pivot_table('prop', rows='year', ...: cols='sex', aggfunc=sum)In [391]: table.plot(title='Sum of table1000.prop by year and sex', ...: yticks=np.linspace(0, 1.2, 13), xticks=range(1880, 2020, 10))

结果如图2-6所示。从图中可以看出，名字的多样性确实出现了增长（前1000项的比例降低）。另一个办法是计算占总出生人数前50%的不同名字的数量，这个数字不太好计算。我们只考虑2010年男孩的名字：

In [392]: df = boys[boys.year == 2010]In [393]: dfOut[393]:Int64Index: 1000 entries, 260877 to 261876Data columns:name 1000 non-null valuessex 1000 non-null valuesbirths 1000 non-null valuesyear 1000 non-null valuesprop 1000 non-null valuesdtypes: float64(1), int64(2), object(2)

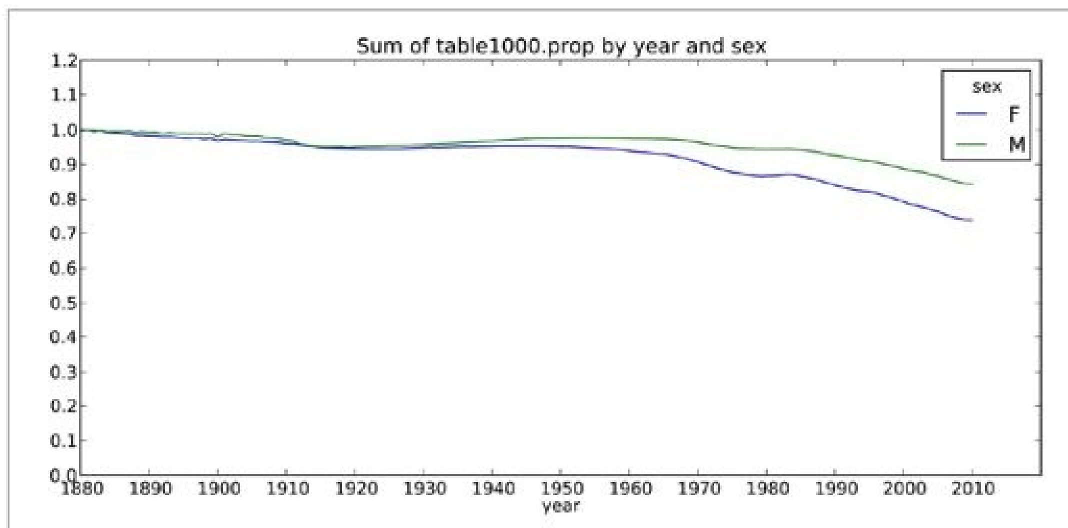


图2-6：分性别统计的前1000个名字在总出生人数中的比例

在对prop降序排列之后，我们想知道前面多少个名字的人数加起来才够50%。虽然编写一个for循环确实也能达到目的，但NumPy有一种更聪明的矢量方式。先计算prop的累计和cumsum，然后再通过searchsorted方法找出0.5应该被插入在哪个位置才能保证不破坏顺序：

In [394]: prop_cumsum = df.sort_index(by='prop', ascending=False).prop.cumsum()In [395]: prop_cumsum[:10]Out[395]:260877 0.011523260878 0.020934260879 0.029959260880 0.038930260881 0.047817260882 0.056579260883 0.065155260884 0.073414260885 0.081528260886 0.089621In [396]: prop_cumsum.searchsorted(0.5)Out[396]: 116

由于数组索引是从0开始的，因此我们要给这个结果加1，即最终结果为117。拿1900年的数据来做个比较，这个数字要小得多：

In [397]: df = boys[boys.year == 1900]In [398]: in1900 = df.sort_index(by='prop', ascending=False).prop.cumsum()In [399]: in1900.searchsorted(0.5) + 1Out[399]: 25

现在就可以对所有year/sex组合执行这个计算了。按这两个字段进行groupby处理，然后用一个函数计算各分组的这个值：

def get_quantile_count(group, q=0.5): group = group.sort_index(by='prop', ascending=False) return group.prop.cumsum().searchsorted(q) + 1diversity = top1000.groupby(['year', 'sex']).apply(get_quantile_count)diversity = diversity.unstack('sex')

现在，diversity这个DataFrame拥有两个时间序列（每个性别各一个，按年度索引）。通过IPython，你可以查看其内容，还可以像之前那样绘制图表（如图2-7所示）：

In [401]: diversity.head()Out[401]:sex F Myear1880 38 141881 38 141882 38 151883 39 151884 39 16In [402]: diversity.plot(title="Number of popular names in top 50%")

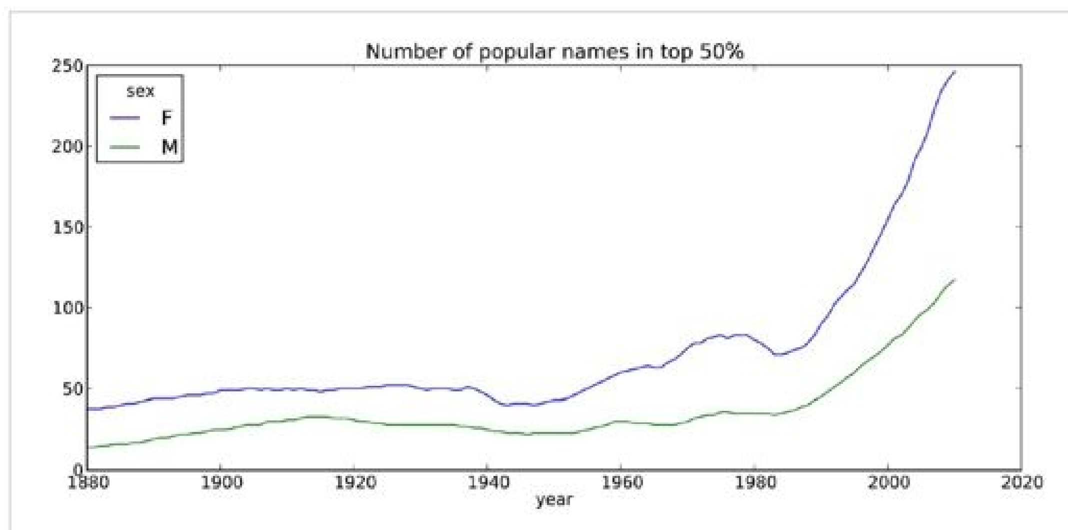


图2-7：按年度统计的密度表

从图中可以看出，女孩名字的多样性总是比男孩的高，而且还在变得越来越高。读者们可以自己分析一下具体是什么在驱动这个多样性（比如拼写形式的变化）。

“最后一个字母”的变革

2007年，一名婴儿姓名研究人员Laura Wattenberg在她自己的网站上指出（http://www.babynamewizard.com）：近百年来，男孩名字在最后一个字母上的分布发生了显著的变化。为了了解具体的情况，我首先将全部出生数据在年度、性别以及末字母上进行了聚合：

从name列取出最后一个字母get_last_letter = lambda x: x[-1]last_letters = names.name.map(get_last_letter)last_letters.name = 'last_letter'table = names.pivot_table('births', rows=last_letters, cols=['sex', 'year'], aggfunc=sum)

然后，我选出具有一定代表性的三年，并输出前面几行：

In [404]: subtable = table.reindex(columns=[1910, 1960, 2010], level='year')In [405]: subtable.head()Out[405]:sex F Myear 1910 1960 2010 1910 1960 2010last_lettera 108376 691247 670605 977 5204 28438b NaN 694 450 411 3912 38859c 5 49 946 482 15476

23125d 6750 3729 2607 22111 262112 44398e 133569 435013 313833 28655 178823 129012

接下来我们需要按总出生数对该表进行规范化处理，以便计算出各性别各末字母占总出生人数的比例：

In [406]: subtable.sum()Out[406]:sex yearF 1910 396416 1960 2022062 2010 1759010M 1910 194198 1960 2132588 2010 1898382In [407]: letter_prop = subtable / subtable.sum().astype(float)

有了这个字母比例数据之后，就可以生成一张各年度各性别的条形图了，如图2-8所示：

import matplotlib.pyplot as pltfig, axes = plt.subplots(2, 1, figsize=(10, 8))letter_prop['M'].plot(kind='bar', rot=0, ax=axes[0], title='Male')letter_prop['F'].plot(kind='bar', rot=0, ax=axes[1], title='Female', legend=False)

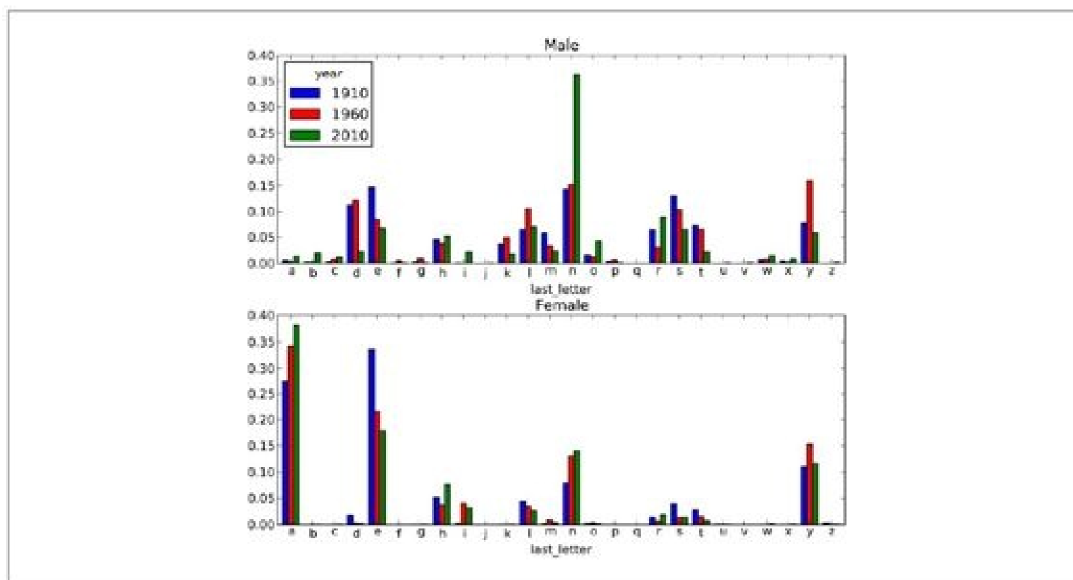


图2-8：男孩女孩名字中各个末字母的比例

从图2-8中可以看出，从20世纪60年代开始，以字母"n"结尾的男孩名字出现了显著的增长。回到之前创建的那个完整表，按年度和性别对其进行规范化处理，并在男孩名字中选取几个字母，最后进行转置以便将各个列做成一个时间序列：

In [410]: letter_prop = table / table.sum().astype(float)In [411]: dny_ts = letter_prop.ix[['d', 'n', 'y'], 'M'].TIn [412]: dny_ts.head()Out[412]: d n yyear1880 0.083055 0.153213 0.0757601881 0.083247 0.153214 0.0774511882 0.085340 0.149560 0.0775371883 0.084066 0.151646 0.0791441884 0.086120 0.149915 0.080405

有了这个时间序列的DataFrame之后，就可以通过其plot方法绘制出一张趋势图了（如图2-9所示）：

In [414]: dny_ts.plot()

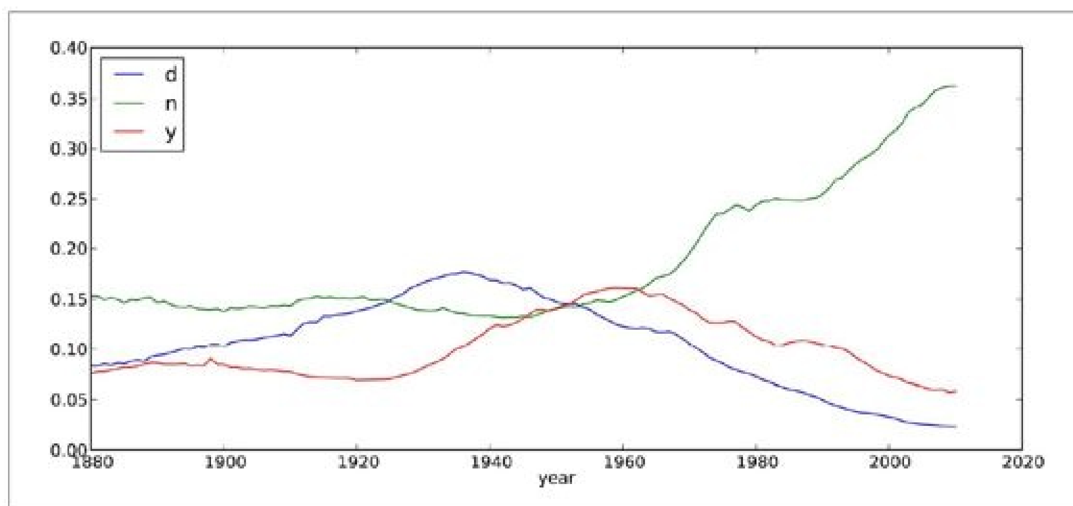


图2-9：各年出生的男孩中名字以d/n/y结尾的人数比例

变成女孩名字的男孩名字（以及相反的情况）

另一个有趣的趋势是，早年流行于男孩的名字近年来“变性了”，例如Lesley或Leslie。回到top1000数据集，找出其中以"lesl"开头的一组名字：

```
In [415]: all_names = top1000.name.unique()In [416]: mask = np.array(['lesl' in x.lower() for x in all_names])In [417]: lesley_like = all_names[mask]In [418]: lesley_likeOut[418]: array([Leslie, Lesley, Leslee, Lesli, Lesly], dtype=object)
```

然后利用这个结果过滤其他的名字，并按名字分组计算出生数以查看相对频率：

```
In [419]: filtered = top1000[top1000.name.isin(lesley_like)]In [420]: filtered.groupby('name').births.sum()Out[420]:nameLeslee 1082Lesley 35022Lesli 929Leslie 370429Lesly 10067Name: births
```

接下来，我们按性别和年度进行聚合，并按年度进行规范化处理：

```
In [421]: table = filtered.pivot_table('births', rows='year', ...: cols='sex', aggfunc='sum')In [422]: table = table.div(table.sum(1), axis=0)In [423]: table.tail()Out[423]:sex F Myear2006 1 NaN2007 1 NaN2008 1 NaN2009 1 NaN2010 1 NaN
```

现在，我们就可以轻松绘制一张分性别的年度曲线图了（如图2-10所示）：

```
In [425]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

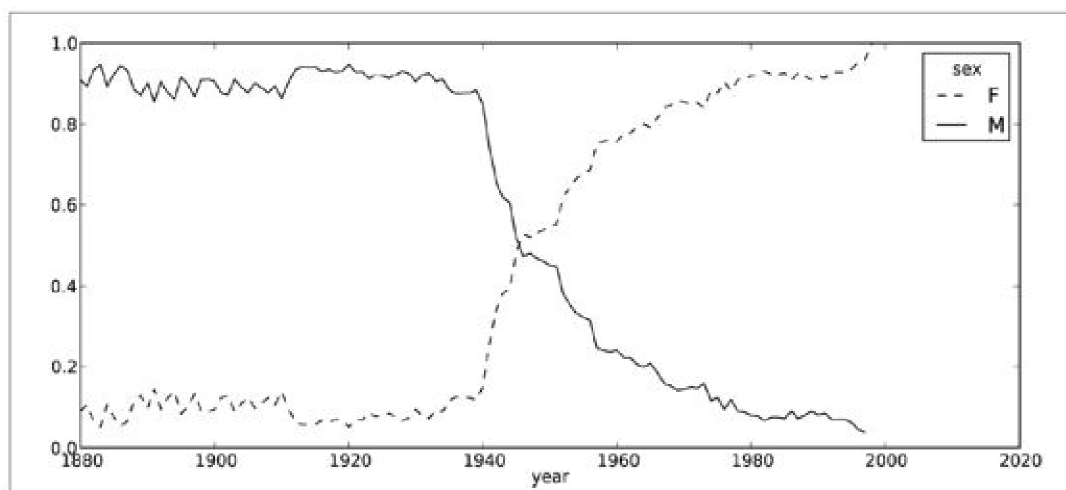


图2-10：各年度使用“Lesley型”名字的男女比例

译注6

：如下链接可能不可用，读者可直接在本书的github上下载。

小结及展望

本章中的这些例子都非常简单，但它们可以让你大致了解后续章节的相关内容。本书关注的焦点是工具而不是那些复杂精妙的分析方法。掌握本书所介绍的技术将使你能够立马开展自己的分析工作（假设你已经知道要做什么了）。

第3章　IPython：一种交互式计算和开发环境

为无为，事无事，味无味。大小多少。报怨以德。

图难于其易，为大于其细；

天下难事，必作于易；天下大事，必作于细。

——老子

人们经常问我，“你的Python开发环境是什么？”我的回答基本永远都是“IPython外加一个文本编辑器”。如果想要得到更加高级的图形化工具和代码自动完成功能，你也可以考虑用一款集成开发环境（IDE）来代替文本编辑器。即便如此，我仍然强烈建议你将IPython作为工作中的重要工具。有的IDE甚至本身就集成了IPython，所以说两全其美的办法还是有的。

2001年，Fernando Pérez为了得到一个更为高效的交互式Python解释器而启动了一个业余项目，于是IPython项目诞生了。在接下来的11年中，它逐渐被公认为现代科学计算中最重要的Python工具之一。IPython本身并没有提供任何的计算或数据分析功能，其设计目的是在交互式计算和软件开发这两个方面最大化地提高生产力。它鼓励一种“执行－探索”（execute explore）的工作模式，而不是许多其他编程语言那种“编辑－编译－运行”（edit-complie-run）的传统工作模式。此外，它跟操作系统shell和文件系统之间也有着非常紧密的集成。由于大部分的数据分析代码都含有探索式操作（试误法和迭代法），因此IPython（在绝大多数情况下）将有助于提高你的工作效率。

当然了，IPython项目现在已经不再只是一个加强版的交互式Python shell，它还有一个可以直接进行绘图操作的GUI控制台、一个基于Web的交互式笔记本，以及一个轻量级的快速并行计算引擎。此外，跟许多其他专为程序员设计（以及由程序员设计）的工具一样，它也是高度可定制的。我将在本章最后介绍一些这样的功能。

由于IPython的核心功能是交互，所以在没有动态控制台的情况下，本章中的某些功能很难说得清楚。如果这是你第一次学习IPython，那我建议你照着例子实际动手试试，感觉一下到底是怎么一回事。跟所有由键盘驱动的控制台环境一样，锻炼对常用命令的肌肉记忆是学习曲线中不可或缺的一部分。

注意： 在第一次阅读时，本章的许多内容都可跳过不看，因为它们对理解本书其余的内容没有影响。本章的目的是让你对IPython所提供的功能有一个全面的了解。

IPython基础

你可以通过命令行启动IPython，就像启动标准Python解释器那样，只是把命令改为ipython罢了：

$ ipythonPython 2.7.2 (default, May 27 2012, 21:26:12)Type "copyright", "credits" or "license" for more information.IPython 0.12 -- An enhanced Interactive Python.? -> Introduction and overview of IPython's features.%quickref -> Quick reference.help -> Python's own help system.object? -> Details about 'object', use 'object??' for extra details.In [1]: a = 5In [2]: aOut[2]: 5

你可以在这里执行任何Python语句，只需将其输入然后按下回车键就行了。如果只是在IPython中输入了一个变量，那么它将会显示出该对象的一个字符串表示：译注1

In[541]：import numpy as npIn[542]: data = {i : randn() for i in range(7)}?In [543]: dataOut[543]:{0: 0.5580886709219381, 1: 0.25701015249982423, 2: 0.8876099192477288, 3: 1.0210657329557034, 4: -0.21799201419817044, 5: 1.1388001234975833, 6: -0.5209890532927175}

许多Python对象都被格式化为可读性更好（或者说排版更好）的形式，这跟print的普通输出形式有着显著区别。如果在标准Python解释器中打印上面那个字典对象，其可读性就没那么好了：

> > > from numpy.random import randn>>> data = {i : randn() for i in range(7)}>>> print data{0: -1.5948255432744511, 1: 0.10569006472787983, 2: 1.972367135977295,3: 0.15455217573074576, 4: -0.24058577449429575, 5: -1.2904897053651216,6: 0.3308507317325902}

IPython还可以方便地执行任意代码块（通过少量优雅的复制粘贴操作）和整个Python脚本。稍后就会对该功能进行介绍。

Tab键自动完成

从表面上看，IPython就像是一个化了淡妆的交互式Python解释器。数学软件（Mathematica）用户可能会对标号式的输入输出提示符眼熟。Tab键自动完成功能是对标准Python shell的主要改进之一，大部分交互式数据分析环境都有这个功能。在shell中输入表达式时，只要按下Tab键，当前命名空间中任何与已输入的字符串相匹配的变量（对象、函数等）就会被找出来：

In [1]: an_apple = 27In [2]: an_example = 42In [3]:an<Tab>译注2an_apple and an_example any译注3

在这个例子中可以看到，IPython将我定义的两个变量都显示出来了，此外还显示了Python关键字and和内置函数any。当然，你也可以在任何对象后面输入一个句点以便自动完成方法和属性的输入：

In [3]: b = [1, 2, 3]In [4]: b.<Tab>b.append b.extend b.insert b.remove b.sortb.count b.index b.pop b.reverse

还可以应用在模块上：

In [1]: import datetimeIn [2]: datetime.<Tab>datetime.MAXYEAR datetime.datetime datetime.timedeltadatetime.MINYEAR datetime.datetime_CAPI datetime.tzinfodatetime.date datetime.time

注意： IPython默认会隐藏那些以下划线开头的方法和属性，比如魔术方法（magic method）以及内部的“私有”方法和属性，其目的是避免在屏幕上显示一堆乱七八糟的东西（也为了避免把Python新人搞晕！）。其实这些也是可以通过Tab键自动完成的，只是你得先输入一个下划线才行。如果你就是喜欢能总是看到这些方法，直接修改IPython配置文件中的相关设置就可以了。

Tab键自动完成功能不只可以用于搜索命名空间和自动完成对象或模块属性。当你输入任何看上去像是文件路径的东西时（即使是在一个Python字符串中），按下Tab键即可找出电脑文件系统中与之匹配的东西：

In [3]: book_scripts/<Tab>译注4book_scripts/cprof_example.py book_scripts/ipython_script_test.pybook_scripts/ipython_bug.py book_scripts/prof_mod.pyIn [3]: path = 'book_scripts/<Tab>book_scripts/cprof_example.py book_scripts/ipython_script_test.pybook_scripts/ipython_bug.py book_scripts/prof_mod.py

如果再结合%run命令（参见后面内容），该功能将显著减少你敲键盘的次数。

Tab键自动完成功能还可用于函数关键字参数（包括等号（=）！）。

译注1

：从输入输出提示符来看，作者在这两段之间做了不少事情，所以如果出现randn找不到的情况，请先执行from numpy.random import randn。

译注2

：后面的<Tab>只是一个按键说明而已，不用输入。顺便说明一下，按完Tab键之后，已输入的内容会在下一行重复出现，行号也是一样的，直接接着往下输入就行了。

译注3

：根据软件版本、配置以及当前上下文的不同，这里得到的结果可能会比书上的多。

译注4

：注意，要使用正斜杠（/），不然认不出来。此外，文件夹或文件名中间不能有空格，不然也无法正常继续操作。

内省

在变量的前面或后面加上一个问号（?）就可以将有关该对象的一些通用信息显示出来：

In [545]: b?Type: listString Form:[1, 2, 3]Length: 3Docstring:list() -> new empty listlist(iterable) -> new list initialized from iterable's items

这就叫做对象内省（object introspection）译注5。如果该对象是一个函数或实例方法，则其docstring（如果有的话）也会被显示出来：

def add_numbers(a, b): """ Add two numbers together Returns ------- the_sum : type of arguments """ return a + b

然后可以利用?来显示这段docstring：

In [547]: add_numbers?Type: functionString Form:File: book_scripts/Definition: add_numbers(a, b)Docstring:Add two numbers togetherReturns-------the_sum : type of arguments

使用??还将显示出该函数的源代码（如果可能的话）：

In [548]: add_numbers??Type: functionString Form:File: book_scripts/Definition: add_numbers(a, b)Source:def add_numbers(a, b): """ Add two numbers together Returns ------- the_sum : type of arguments """ return a + b

?还有一个用法，即搜索IPython命名空间，类似于标准UNIX或Windows命令行中的那种用法。一些字符再配以通配符（*）即可显示出所有与该通配符表达式相匹配的名称。例如，我们可以列出NumPy顶级命名空间中含有"load"的所有函数：

In [549]: np.*load*?np.loadnp.loadsnp.loadtxtnp.pkgload

%run命令

在IPython会话环境中，所有文件都可以通过%run命令当做Python程序来运行。假设你在ipython_script_test.py中存放了一段简单的脚本，如下所示：

def f(x, y, z): return (x + y) / za = 5b = 6c = 7.5result = f(a, b, c)

只要将文件名传给%run就可以运行了：

In [550]: %run ipython_script_test.py译注6

脚本是在一个空的命名空间中运行的（没有任何import，也没有定义任何其他的变量），所以其行为应该跟在标准命令行环境（通过python script.py启动的）中执行时一样。此后，该文件中所定义的全部变量（还有各种import、函数和全局变量）就可以在当前IPython shell中访问了（除非发生了异常）：

In [551]: cOut[551]: 7.5In [552]: resultOut[552]: 1.4666666666666666

如果Python脚本需要用到命令行参数（通过sys.argv访问），可以将参数放到文件路径的后面，就像在命令行上执行那样。

注意：如果希望脚本能够访问在交互式IPython命名空间译注7中定义的变量，那就应该使用%run i而不是%run。

中断正在执行的代码

任何代码在执行时（无论是通过%run执行的脚本，还是长时间运行的命令），只要按下"Ctrl-C"，就会引发一个KeyboardInterrupt。除一些非常特殊的情况之外，绝大部分Python程序都将立即停止执行。

警告：当Python代码已经调用了某个已编译的扩展模块时，按下"Ctrl-C"将无法使程序立即停止执行。在这种情况下，要么只能等待Python解释器重新获得控制权，要么只能通过操作系统的任务管理器强制终止Python进程（比较极端的情况下才需要这么干）。

执行剪贴板中的代码

在IPython中执行代码的最简单方式是粘贴剪贴板中的代码。虽然这种做法很粗糙，但在实际工作中却很有用。比如说，在开发一个复杂或费时的应用程序时，你可能希望能一段一段地执行脚本，以便查看各个阶段所加载的数据以及产生的结果。又比如说，你在网上找了一段合用的代码，但又不想专门为其新建一个.py文件。

多数情况下，我们都可以通过"Ctrl-Shift-V"将剪贴板中的代码片段粘贴出来译注8。注意，这并不是万试万灵的，因为这种粘贴方式模拟的是在IPython中逐行输入代码，换行符会被处理为<return>。也就是说，如果你所粘贴的是一段缩进代码，且其中有一个空行，IPython就会认为缩进在空行那里结束了。当执行到缩进块后面那行代码时，就会引发一个IndentationError。例如下面这段代码：

x = 5y = 7if x > 5: x += 1   y = 8

直接粘贴是不行的：

In [1]: x = 5In [2]: y = 7In [3]: if x > 5: ...: x += 1 ...:In [4]: y = 8IndentationError: unexpected indentIf you want to paste code into IPython, try the %paste and %cpaste magic functions.

正如错误提示信息所说的那样，我们应该使用%paste和%cpaste这两个魔术函数。%paste可以承载剪贴板中的一切文本译注9，并在shell中以整体形式执行译注10：

In [6]: %pastex = 5y = 7if x > 5: x += 1   y = 8## -- End pasted text --

警告：根据你的系统平台以及Python的安装情况，%paste可能会不起作用。EPDFree（在第1章中介绍过）等打包发布的版本应该没有问题。

%cpaste跟%paste差不多译注11，只不过它多出了一个用于粘贴代码的特殊提示符而已：

In [7]: %cpastePasting code; enter '--' alone on the line to stop or use Ctrl-D.:x = 5:y = 7:if x > 5:: x += 1:: y = 8:--

对于%cpaste块，在最终执行之前，你想粘贴多少代码就粘贴多少。如果想在执行那些粘贴进去的代码之前先检查一番，就可以考虑使用%cpaste。如果发现粘贴的代码有错，只需按下"Ctrl-C"即可终止%cpaste提示符。

后面我将会介绍IPython HTML Notebook，它使我们能以一种基于浏览器的notebook格式逐段对可执行代码单元进行分析。

IPython跟编辑器和IDE之间的交互

某些文本编辑器（如Emacs和vim）带有一些能将代码块直接发送到IPython shell的第三方扩展。详情请参考IPython网站或搜索引擎。

某些IDE（如PyDev plugin for Eclipse和Python Tools for Visual Studio（微软出品））都集成了IPython终端应用程序。如果你既想用IDE又不想放弃IPython控制台，这可能是个不错的选择。

键盘快捷键

IPython提供了许多用于提示符导航（Emacs文本编辑器或UNIX bash shell的用户对此会很熟悉）和查阅历史shell命令（详见下一节）的键盘快捷键。表3-1总结了最常用的一些快捷键。图3-1说明了几个光标移动快捷键的功能。

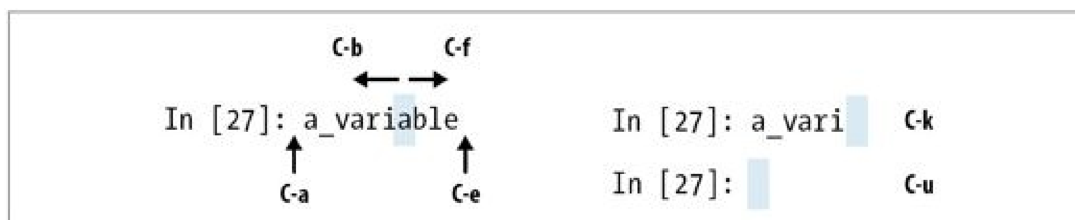


图3-1：几个IPython键盘快捷键的用法

表3-1：IPython标准键盘快捷键

命令	说明
Ctrl-P或上箭头键	后向搜索命令历史中以当前输入的文本开头的命令
Ctrl-N或下箭头键	前向搜索命令历史中以当前输入的文本开头的命令
Ctrl-R	按行读取的反向历史搜索（部分匹配）
Ctrl-Shift-v	从剪贴板粘贴文本
Ctrl-C	中止当前正在执行的代码
Ctrl-A	将光标移动到行首
Ctrl-E	将光标移动到行尾
Ctrl-K	删除从光标开始至行尾的文本
Ctrl-U	清除当前行的所有文本[译注12]
Ctrl-F	将光标向前移动一个字符
Ctrl-b	将光标向后移动一个字符
Ctrl-L	清屏

译注12：这个快捷键的功能只是跟Ctrl-K相反而已，即删除从光标开始至行首的文本，并非完全删除。

异常和跟踪

如果%run某段脚本或执行某条语句时发生了异常，IPython默认会输出整个调用栈跟踪（traceback），其中还会附上调用栈各点附近的几行代码作为上下文参考。

In [553]: %run ch03/ipythonbug.py---AssertionError Traceback (most recent call last)/home/wesm/code/ipython/IPython/utils/py3compat.pyc in execfile(fname, *where) 176 else: 177 filename = fname--> 178 _builtin.execfile(filename, *where)book_scripts/ch03/ipython_bug.py in <module>() 13 throws_an_exception() 14---> 15 calling_things()book_scripts/ch03/ipython_bug.py in calling_things() 11 def calling_things(): 12 works_fine()---> 13 throws_an_exception() 14 15 calling_things()book_scripts/ch03/ipython_bug.py in throws_an_exception() 7 a = 5 8 b = 6----> 9 assert(a + b == 10) 10 11 def calling_things():AssertionError:

拥有额外的上下文代码参考是它相对于标准Python解释器的一大优势。上下文代码参考的数量可以通过%xmode魔术命令进行控制，既可以少（与标准Python解释器相同）也可以多（带有函数参数值以及其他信息）。本章稍后还会讲到如何在发生异常之后进入跟踪栈进行交互式的事后调试（post-mortem debugging）。

魔术命令

IPython有一些特殊命令（被称为魔术命令（Magic Command）），它们有的为常见任务提供便利，有的则使你能够轻松控制IPython系统的行为。魔术命令是以百分号%为前缀的命令。例如，你可以通过%timeit这个魔术命令检测任意Python语句（如矩阵乘法）的执行时间

（稍后将对此进行详细讲解）：

In [554]: a = np.random.randn(100, 100)In [555]: %timeit np.dot(a, a)10000 loops, best of 3: 69.1 us per loop

魔术命令可以看做运行于IPython系统中的命令行程序。它们大都还有一些“命令行选项”，使用？即可查看其选项：

In [1]: %reset?Resets the namespace by removing all names defined by the user.Parameters---------- -f : force reset without asking for confirmation. -s : 'Soft' reset: Only clears your namespace, leaving history intact. References to objects may be kept. By default (without this option), we do a 'hard' reset, giving you a new session and removing all references to objects from the current session.Examples--------In [6]: a = 1In [7]: aOut[7]: 1In [8]: 'a' in _ip.user_nsOut[8]: TrueIn [9]: %reset -fIn [1]: 'a' in _ip.user_nsOut[1]: False

魔术命令默认是可以不带百分号使用的，只要没有定义与其同名的变量即可。这个技术叫做automagic，可以通过%automagic打开或关闭。

由于可以在IPython系统中直接访问它的文档，因此我建议你浏览一下所有这些特殊的命令（输入%quickref或%magic即可）。我将着重讲解几个重要的有助于交互式计算和Python开发的魔术命令。

表3-2：常用的IPython魔术命令

命令	说明
%quickref	显示IPython的快速参考
%magic	显示所有魔术命令的详细文档
%debug	从最新的异常跟踪的底部进入交互式调试器
%hist	打印命令的输入（可选输出）历史
%pdb	在异常发生后自动进入调试器
%paste	执行剪贴板中的Python代码

表3-2：常用的IPython魔术命令（续）

命令	说明
%cpaste	打开一个特殊提示符以便手工粘贴待执行的Python代码
%reset	删除interactive命名空间中的全部变量/名称
%page *OBJECT*	通过分页器打印输出OBJECT
%run *script.py*	在IPython中执行一个Python脚本文件
%prun *statement*	通过cProfile执行statement，并打印分析器的输出结果
%time *statement*	报告statement的执行时间
%timeit *statement*	多次执行statement以计算系综平均执行时间。对那些执行时间非常小的代码很有用
%who、%who_ls、%whos	显示interactive命名空间中定义的变量，信息级别/冗余度可变
%xdel *variable*	删除variable，并尝试清除其在IPython中的对象上的一切引用

基于Qt的富GUI控制台

IPython团队开发了一个基于Qt框架（其目的是为终端应用程序提供诸如内嵌图片、多行编辑、语法高亮之类的富文本编辑功能）的GUI控制台（见图3-2）。如果你已经安装了PyQt或PySide，使用下面这条命令来启动的话即可为其添加绘图功能：

ipython qtconsole --pylab=inline

Qt控制台可以通过标签页的形式启动多个IPython进程，这就使你能够在多个任务之间轻松切换。它也可以跟IPython HTML Notebook应用程序共享同一个进程，稍后我将专门对此进行讲解。

matplotlib集成与pylab模式

导致IPython广泛应用于科学计算领域的部分原因是它能跟matplotlib这样的库以及其他GUI工具集默契配合。即使你从未使用过matplotlib也不用担心，本书稍后会对其进行详细讲解。如果在标准Python shell中创建一个matplotlib绘图窗口，你就会郁闷地发现，GUI的事件循环会接管Python会话的控制权，直到该绘图窗口关闭为止。这自然无法实现交互式的数据分析和可视化，因此IPython对各个GUI框架进行了专门的处理以使其能够跟shell配合得天衣无缝。

通常，我们通过在启动IPython时加上--pylab（注意是两个短划线）标记来集成matplotlib（见图3-3）。

$ ipython --pylab

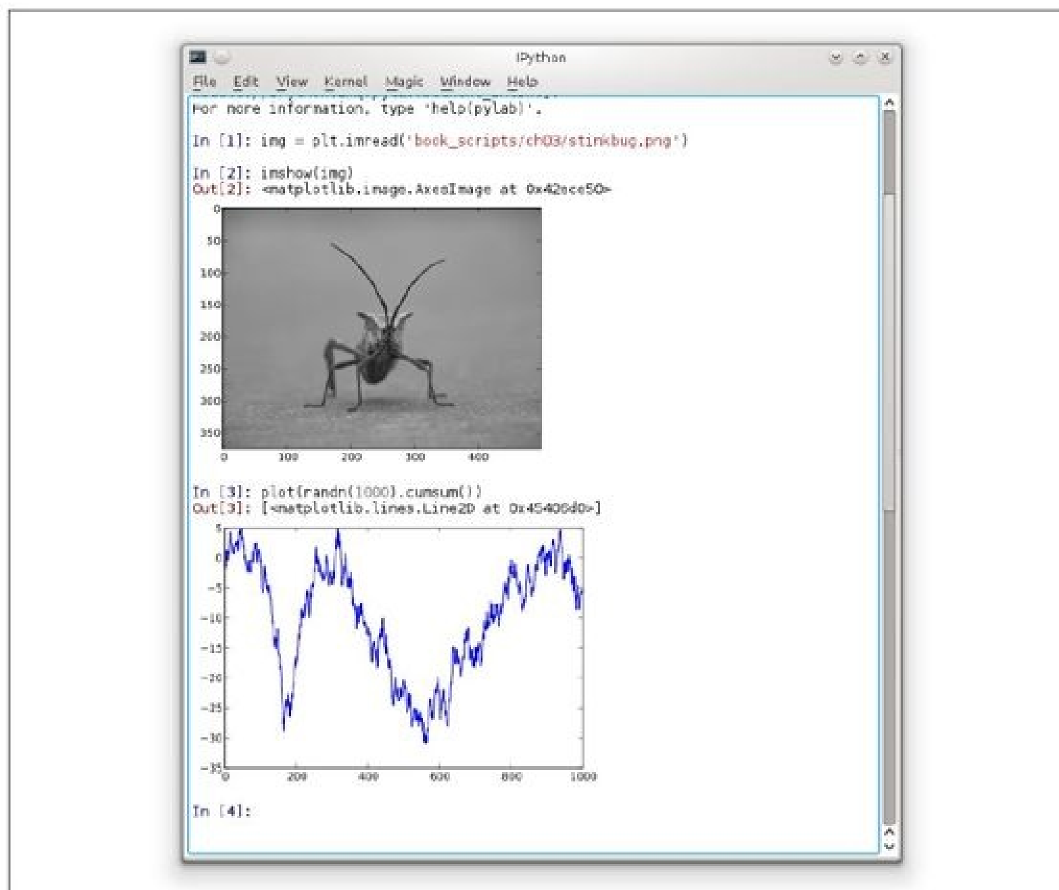


图3-2：IPython的Qt控制台

这样会导致几个结果。第一，IPython会启用默认GUI后台集成，这样matplotlib绘图窗口的创建就没问题了。第二，NumPy和matplotlib的大部分功能会被引入到最顶层的interactive命名空间以产生一个交互式的计算环境（就像MATLAB和其他领域特定型科学计算环境那样）。也可以通过%gui对此进行手工设置（详情请执行%gui?）。

译注5

：也有译作内视、自省的，不过更多译作内省。

译注6

：注意文件的路径，这里实际上用的是默认路径。简单一点的办法就是直接写绝对路径，肯定不出错。

译注7

：该命名空间的名字就是interactive。

译注8

：Windows中此法行不通，需要用右键菜单中的粘贴功能，否则仅显示第一行。

译注9

：注意，这里说的是“一切”。

译注10

：注意，由于是立即整体执行，所以不要复制%paste。没事干的话倒是可以试试。

译注11

：建议始终用这个，虽然稍微麻烦一点，但是出错的可能性小很多。

使用命令历史

IPython维护着一个位于硬盘上的小型数据库，其中含有你执行过的每条命令的文本。这样做有几个目的：

·只需很少的按键次数即可搜索、自动完成并执行之前已经执行过的命令。

·在会话间持久化命令历史。

·将输入/输出历史记录到日志文件。

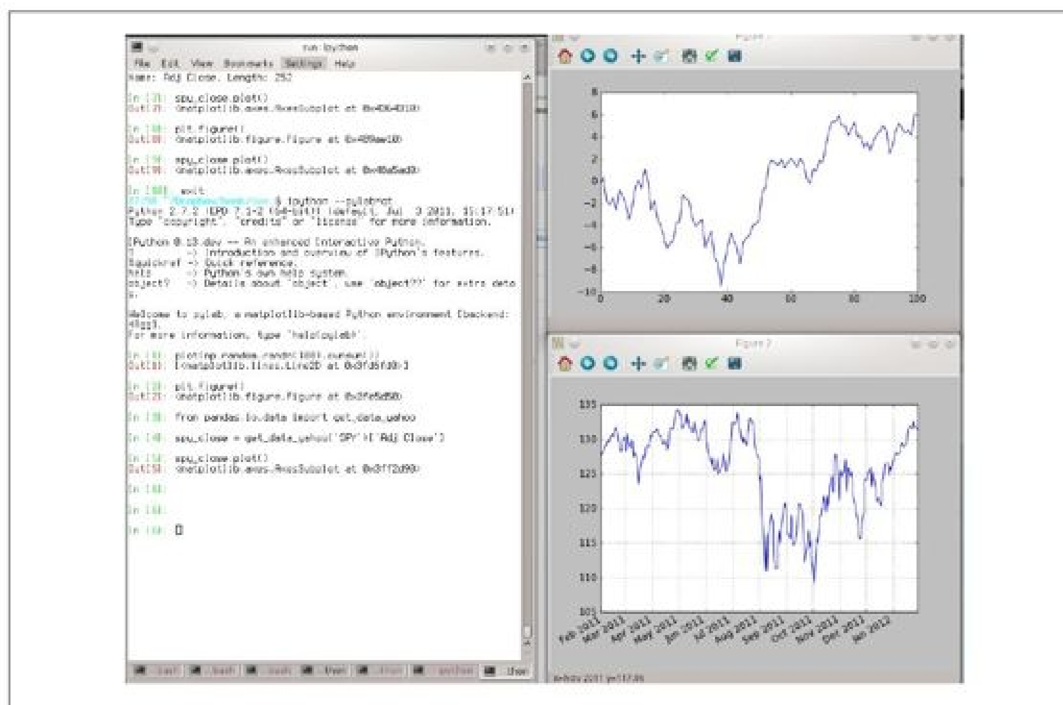


图3-3：pylab模式：IPython和matplotlib窗口

搜索并重用命令历史

对于许多人来说，能够搜索并执行前面的命令是非常有用的功能。IPython倡导的是一种迭代的、交互式的开发模式：你可能常常会发现自己总是在重复输入相同的命令（比如%run命令或其他的代码片段）。假设你已经执行了：

In[7]: %run first/second/third/data_script.py

而在查看其执行结果后（假设其已经成功执行完毕）发现计算过程不对。在找出问题原因并修改了data_script.py之后，只需输入%run命令的前几个字符并按"Ctrl-P"键或上箭头键即可。这样就会搜索出命令历史中第一个与你输入的字符相匹配的命令。多次按"Ctrl-P"键或上箭头键就会在命令历史中不断搜索。如果你错过了想要的那条命令也没关系，你可以按"Ctrl-N"键或下箭头键在命令历史中前向搜索。只要多操作几次，以后你会想都不想地按下这些键！

"Ctrl-R"用于实现部分增量搜索，跟UNIX型shell中的readline所提供的功能一样。在Windows上，IPython模拟了readline功能。按下"Ctrl-R"并输入你想搜索的行中的几个字符：

In [1]: a_command = foo(x, y, z)(reverse-i-search)`com': a_command = foo(x, y, z)

按下"Ctrl-R"将会循环搜索命令历史中每一条与输入相符的行。

输入和输出变量

忘记把函数结果赋值给变量是一件让人很郁闷的事情。好在IPython会将输入（你输入的那些文本）和输出（返回的对象）的引用保存在一些特殊变量中。最近的两个输出结果分别保存在（一个下划线）和_（两个下划线）变量中：

In [556]: 2 ** 27Out[556]: 134217728In [557]: _Out[557]: 134217728

输入的文本被保存在名为_iX的变量中，其中X是输入行的行号。每个输入变量都有一个对应的输出变量_X。比如说，在输入完第27行后，就会产生两个新变量_27（输出变量）和_i27（输入变量）。

In [26]: foo = 'bar'In [27]: fooOut[27]: 'bar'In [28]: _i27Out[28]: u'foo'In [29]: _27Out[29]: 'bar'

由于输入变量是字符串，因此可以用Python的exec关键字重新执行：

In [30]: exec _i27

有几个魔术命令可用于控制输入和输出历史。%hist用于打印全部或部分输入历史，可以选择是否带行号。%reset用于清空interactive命名空间，并可选择是否清空输入和输出缓存。%xdel用于从IPython系统中移除特定对象的一切引用。详细信息请参考相应魔术命令的文档。

警告： 在处理非常大的数据集时，一定要注意IPython的输入输出历史，它会导致所有对象引用都无法被垃圾收集器处理（即释放内存），即使用del关键字将变量从interactive命名空间中删除也不行。对于这种情况，谨慎地使用%xdel和%reset将有助于避免出现内存方面的问题。

记录输入和输出

IPython能够记录整个控制台会话，包括输入和输出。执行%logstart即可开始记录日志：

In [3]: %logstartActivating auto-logging. Current session state plus future input saved.Filename : ipython_log.pyMode : rotateOutput logging : FalseRaw input log : FalseTimestamping : FalseState : active

IPython的日志功能可以在任何时刻开启，它将记录你的整个会话（包括此前的命令）。因此，如果你在写代码的过程中，突然想要保存所有工作的时候，直接启动日志功能就行了。%logstart的具体选项（比如修改输出文件路径）请参考其文档，此外还可以看看几个与之配套的魔术命令%logoff、%logon、%logstate以及%logstop。

与操作系统交互

IPython的另一个重要特点就是它跟操作系统shell结合得非常紧密。也就是说，你可以直接在IPython中实现标准的Windows或UNIX（Linux、OS X）命令行活动。比如执行shell命令、更改目录、将命令的执行结果保存在Python对象（列表或字符串）中等。此外，它还提供了shell命令别名以及目录书签等功能。

表3-3总结了用于调用shell命令的魔术命令及其语法。我将在后面几节中简要介绍这些功能。

表3-3：跟系统相关的IPython魔术命令

命令	说明
!cmd	在系统shell中执行cmd
output = !cmd args	执行cmd，并将stdout存放在output中
%alias *alias_name cmd*	为系统shell命令定义别名
%bookmark	使用IPython的目录书签系统
%cd *directory*	将系统工作目录更改为directory
%pwd	返回系统的当前工作目录
%pushd *directory*	将当前目录入栈，并转向目标目录
%popd	弹出栈顶目录，并转向该目录
%dirs	返回一个含有当前目录栈的列表
%dhist	打印目录访问历史
%env	以dict形式返回系统环境变量

shell命令和别名

在IPython中，以感叹号（!）开头的命令行表示其后的所有内容需要在系统shell中执行。也就是说，你可以删除文件（根据OS的不同，使用rm或del）、修改目录或执行任意其他处理过程。甚至还可以启动一些能将控制权从IPython手中夺走的进程（比如另外再启动一个Python解释器）：

In [2]: !pythonPython 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul 3 2011, 15:17:51)[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)] on linux2Type "packages", "demo" or "enthought" for more information.>>>

此外，还可以将shell命令的控制台输出存放到变量中，只需将！开头的表达式赋值给变量即可。例如，我的Linux电脑通过以太网连接到互联网，于是可以将我的IP地址存到一个Python变量中去：译注13

In [1]: ip_info = !ifconfig eth0 | grep "inet"In [2]: ip_info[0].strip()Out[2]: 'inet addr:192.168.1.137 Bcast:192.168.1.255 Mask:255.255.255.0'

返回的Python对象ip_info实际上是一个含有控制台输出结果的自定义列表类型。

在使用！时，IPython还允许使用当前环境中定义的Python值。只需在变量名前面加上美元符号（$）即可：译注14

In [3]: foo = 'test*'In [4]: !ls $footest4.py test.py test.xml

魔术命令%alias可以为shell命令自定义简称。例如：

In [1]: %alias ll ls -lIn [2]: ll /usrtotal 332drwxr-xr-x 2 root root 69632 2012-01-29 20:36 bin/drwxr-xr-x 2 root root 4096 2010-08-23 12:05 games/drwxr-xr-x 123 root root 20480 2011-12-26 18:08 include/drwxr-xr-x 265 root root 126976 2012-01-29 20:36 lib/drwxr-xr-x 44 root root 69632 2011-12-26 18:08 lib32/lrwxrwxrwx 1 root root 3 2010-08-23 16:02 lib64 -> lib/drwxr-xr-x 15 root root 4096 2011-10-13 19:03 local/drwxr-xr-x 2 root root 12288 2012-01-12 09:32 sbin/drwxr-xr-x 387 root root 12288 2011-11-04 22:53 share/drwxrwsr-x 24 root src 4096 2011-07-17 18:38 src/

可以一次执行多条命令，只需将它们写在一行上并以分号隔开即可：

In [558]: %alias test_alias (cd ch08; ls; cd ..)In [559]: test_aliasmacrodata.csv spx.csv tips.csv

注意，IPython会在会话结束时立即“忘记”你所定义的一切别名。为了创建永久性的别名，你需要使用配置系统。本章稍后会对此进行介绍。

目录书签系统

IPython有一个简单的目录书签系统，它使你能保存常用目录的别名以便实现快速跳转。比如说，作为一名狂热的Dropbox用户，为了能够快速地转到我的Dropbox目录，我可以定义一个书签：

In [6]: %bookmark db /home/wesm/Dropbox/

在定义好书签之后，就可以在执行魔术命令%cd时使用这些书签了：

In [7]: cd db(bookmark:db) -> /home/wesm/Dropbox//home/wesm/Dropbox

如果书签名与当前工作目录中的某个目录名冲突，可以通过-b标记（其作用是覆写）使用书签目录。%bookmark的-l选项的作用是列出所有书签：

In [8]: %bookmark -lCurrent bookmarks:db -> /home/wesm/Dropbox/

书签跟别名的区别在于，它们会被自动持久化。

译注13

：之前已经说过，作者用的不是Windows操作系统，所以这个命令自然无法执行。Windows上可以用ipconfig，但毕竟不是一样东西，这里的代码自己能看明白即可。

译注14

：在Windows中，将ls换成dir。

软件开发工具

IPython不仅是一种舒适的交互式计算和数据分析环境，同时也非常适合成为一种软件开发环境。在数据分析应用程序中，最重要的事情就是拥有正确的代码。幸运的是，IPython紧密集成并加强了Python内置的pdb调试器。此外，你还希望代码运行能足够快。为此，IPython提供了一些简单易用的代码运行时间及性能分析工具。下面，我将对这些工具做一个详细介绍。

交互式调试器

IPython的调试器增强了pdb，如Tab键自动完成、语法高亮、为异常跟踪的每条信息添加上下文参考等。调试代码的最佳时机之一就是错误刚刚发生那会儿。%debug命令（在发生异常之后马上输入）将会调用那个“事后”调试器，并直接跳转到引发异常的那个栈帧（stack frame）：

```
In [2]: run ch03/ipython_bug.py---------------------------------------------------------------------------
AssertionError Traceback (most recent call
last)/home/wesm/book_scripts/ch03/ipython_bug.py in <module>() 13
throws_an_exception() 14---> 15
calling_things()/home/wesm/book_scripts/ch03/ipython_bug.py in calling_things() 11 def
calling_things(): 12 works_fine()---> 13 throws_an_exception() 14 15
calling_things()/home/wesm/book_scripts/ch03/ipython_bug.py in throws_an_exception() 7 a
= 5 8 b = 6----> 9 assert(a + b == 10) 10 11 def calling_things():AssertionError:In [3]:
%debug> /home/wesm/book_scripts/ch03/ipython_bug.py(9)throws_an_exception() 8 b = 6--
--> 9 assert(a + b == 10) 10ipdb>
```

在这个调试器中，你可以执行任意Python代码并查看各个栈帧中的一切对象和数据（也就是解释器还“留了条生路”的那些）。默认是从最低级开始的（即错误发生的地方）。输入u（或up）和d（或down）即可在栈跟踪的各级别之间切换：

```
ipdb> u> /home/wesm/book_scripts/ch03/ipython_bug.py(13)calling_things() 12
works_fine()---> 13 throws_an_exception() 14
```

执行%pdb命令可以让IPython在出现异常之后自动调用调试器。很多人都认为这是一个非常实用的功能。

此外，调试器还可以为代码开发工作提供帮助，尤其是当你想要设置断点或对函数/脚本进行单步调试以查看各条语句的执行情况时。实现这个目的的方式有几个。第一，使用带有-d选项的%run命令，这将会在执行脚本文件中的代码之前先打开调试器。必须立即输入s（或step）才能进入脚本：译注15

```
In [5]: run -d ch03/ipython_bug.pyBreakpoint 1 at
/home/wesm/book_scripts/ch03/ipython_bug.py:1NOTE: Enter 'c' at the ipdb> prompt to
start your script.> <string>(1)<module>()ipdb> s> g:\ipython_bug.py(1)<module>()1---> 1 def
works_fine(): 2 a = 5 3 b = 6
```

在此之后，该文件接下来的执行方式就全凭你一句话了。比如说，在上面那个异常中，我们可以在调用works_fine方法的地方设置一个断点，然后输入c（或continue）使脚本一直运行下去直到该断点时为止：

ipdb> b 12ipdb> c> /home/wesm/book_scripts/ch03/ipython_bug.py(12)calling_things() 11 def calling_things():2--> 12 works_fine() 13 throws_an_exception()

这时可以单步进入works*fine()*或执行*worksfine()*（输入*n*（或*next*）直接执行到下一行*[译注16](#809468440711498-Yi_Zhu_16_Ye_Jiu_Shi_step_over)*）：

ipdb> n> /home/wesm/book_scripts/ch03/ipython_bug.py(13)calling_things()2 12 works_fine()---> 13 throws_an_exception() 14

然后，我们单步进入throws_an_exception并前进到发生错误的那一行，查看在此范围内的变量。注意，调试器命令的优先级高于变量名。这时在变量前面加上感叹号（！）即可查看其内容。

ipdb> s--Call--> /home/wesm/book_scripts/ch03/ipython_bug.py(6)throws_an_exception() 5----> 6 def throws_an_exception(): 7 a = 5ipdb> n> /home/wesm/book_scripts/ch03/ipython_bug.py(7)throws_an_exception() 6 def throws_an_exception():----> 7 a = 5 8 b = 6ipdb> n> /home/wesm/book_scripts/ch03/ipython_bug.py(8)throws_an_exception() 7 a = 5----> 8 b = 6 9 assert(a + b == 10)ipdb> n> /home/wesm/book_scripts/ch03/ipython_bug.py(9)throws_an_exception() 8 b = 6----> 9 assert(a + b == 10) 10ipdb> !a5ipdb> !b6

要想精通这个交互式调试器，必须经过大量的实践才行。表3-4列出了该调试器的全部命令。如果你习惯了使用某款IDE，刚开始用这种终端型调试器的时候可能会觉得有点麻烦，但慢慢就会习惯了。虽然大部分Python IDE都拥有优秀的GUI调试器，但是在IPython中调试程序却往往会带来更高的生产率。

表3-4：(I)Python调试器命令

命令	功能
h(elp)	显示命令列表
help *command*	显示*command*的文档
c(ontinue)	恢复程序的执行

表3-4：(I)Python调试器命令（续）

命令	功能
q(uit)	退出调试器，不再执行任何代码
b(reak) *number*	在当前文件的第*number*行设置一个断点
b path/to/*file.py:number*	在指定文件的第*number*行设置一个断点
s(tep)	单步进入函数调用
n(ext)	执行当前行，并前进到当前级别的下一行
u(p)/d(own)	在函数调用栈中向上或向下移动
a(rgs)	显示当前函数的参数
debug *statement*	在新的（递归）调试器中调用语句*statement*
l(ist) *statement*	显示当前行，以及当前栈级别上的上下文参考代码
w(here)	打印当前位置的完整栈跟踪（包括上下文参考代码）

调试器的其他使用场景

除上面提到的之外，还有另外几种调用调试器的手段。第一，使用set*trace*这个特别的函数（以*pdb.settrace*命名），这差不多可以算作一种"穷人的断点*[*译注*17](#809468440711498-YiZhu_17_Zuo_Zhe_Zai_Zhe_Li_De_Yi_Si_Shi_Zhe_Zhong_Duan_Dian_Bi_Jiao_Sui_Bian__Shi_Ying_Bian_Ma_De*)"。下面这两个方法可能会在你的日常工作中派上用场（你也可以像我一样直接将其添加到IPython配置中）：

def set_trace(): from IPython.core.debugger import Pdb Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)def debug(f, *args, **kwargs): from IPython.core.debugger import Pdb pdb = Pdb(color_scheme='Linux') return pdb.runcall(f,* args, **kwargs)

第一个函数（set_trace）非常简单。你可以将其放在代码中任何希望停下来查看一番的地方（比如发生异常的地方）：

In [7]: run ch03/ipython_bug.py> /home/wesm/book_scripts/ch03/ipython_bug.py(16)calling_things() 15 set_trace()---> 16 throws_an_exception() 17

按下c（或continue）仍然会使代码恢复执行，不受任何影响。

另外那个debug函数使你能够直接在任意函数上使用调试器。假设我们写了如下函数：

def f(x, y, z=1): tmp = x + y return tmp / z

现在想对其进行单步调试。f的正常使用方式应该类似于f(1,2,z=3)这个样子。为了能够单步进入f，将f作为第一个参数传给debug，后面按顺序再跟上各个需要传给f的关键字参数：

In [6]: debug(f, 1, 2, z=3)> (2)f() 1 def f(x, y, z):----> 2 tmp = x + y 3 return tmp / zipdb>

我发现这两个函数虽然简单，但是在日常工作当中却节省了我不少的时间。

此外，这个调试器还可以结合%run使用。通过%run-d执行脚本，你将会直接进入调试器，然后可以设置一些断点并启动脚本：

In [1]: %run -d ch03/ipython_bug.pyBreakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:1NOTE: Enter 'c' at the ipdb> prompt to start your script.> <string>(1)<module>()ipdb>

如果再加上-b和一个行号，则调试器在启动时就会自动设置一个断点：

In [2]: %run -d -b2 ch03/ipython_bug.pyBreakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:2NOTE: Enter 'c' at the ipdb> prompt to start your script.> <string>(1)<module>()ipdb> c> /home/wesm/book_scripts/ch03/ipython_bug.py(2)works_fine() 1 def works_fine():1---> 2 a = 5 3 b = 6ipdb>

测试代码的执行时间：%time和%timeit

对于规模更大、运行时间更长的数据分析应用程序，你可能会希望测试一下各个部分或函数调用或语句的执行时间。你可能会希望了解某个复杂计算过程中到底是哪些函数占用的时间最多。幸运的是，在开发和测试代码的过程中，IPython能够让你轻松得到这些信息。使用内置的time模块及其time.clock和time.time函数手工测试代码执行时间是一件令人烦闷的事情，因为你必须编写许多一模一样的了无生趣的公式化代码：

import timestart = time.time()for i in range(iterations): # 这里放一些待执行的代码 elapsed_per = (time.time() - start) / iterations

由于这是一个非常常用的功能，所以IPython专门提供了两个魔术函数（%time和%timeit）以便自动完成该过程。%time一次执行一条语句，然后报告总体执行时间。假设我们有一大堆字符串，希望对几个“能够选出具有特殊前缀的字符串”的函数进行比较。下面是一个拥有60万字符串的数组，以及两个不同的“能够选出其中以foo开头的字符串”的方法：

一个非常大的字符串数组strings = ['foo', 'foobar', 'baz', 'qux', 'python', 'Guido Van Rossum'] * 100000method1 = [x for x in strings if x.startswith('foo')]method2 = [x for x in strings if x[:3] == 'foo']

看上去它们的性能表现应该差不多，对吧？我们通过%time来确认一下：

In [561]: %time method1 = [x for x in strings if x.startswith('foo')]CPU times: user 0.19 s, sys: 0.00 s, total: 0.19 sWall time: 0.19 sIn [562]: %time method2 = [x for x in strings if x[:3] == 'foo']CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 sWall time: 0.09 s

墙上时间（Wall time）是我们最感兴趣的数字。所以，看上去第一个方法耗费了两倍以上的时间，但这并不是一个非常精确的结果。如果你对相同语句多次执行%time的话，就会发现其结果是会变的。为了得到更为精确的结果，需要使用魔术函数%timeit。对于任意语句，它会自动多次执行以产生一个非常精确的平均执行时间。

In [563]: %timeit [x for x in strings if x.startswith('foo')]10 loops, best of 3: 159 ms per loopIn [564]: %timeit [x for x in strings if x[:3] == 'foo']10 loops, best of 3: 59.3 ms per loop

这个貌似平淡无奇的例子正好说明了一个事实：我们非常有必要了解Python标准库、NumPy、pandas以及本书中所用到的其他库的性能特点。在大型数据分析应用程序中，这些不起眼的毫秒数是会不断累积的！

对于那些执行时间非常短（甚至是那些微秒（1e-6秒）或纳秒（1e-9秒）级的）的分析语句和函数而言，%timeit是非常有用的。虽然这些时间值小到几乎可以忽略不计，但同样执行100万次一个20微秒的函数，所用的时间要比一个5微秒的多15秒。在上面那个例子中，我们可以直接对那两个字符串运算进行比较以了解其性能特点：

In [565]: x = 'foobar'In [566]: y = 'foo'In [567]: %timeit x.startswith(y)1000000 loops, best of 3: 267 ns per loopIn [568]: %timeit x[:3] == y10000000 loops, best of 3: 147 ns per loop

基本性能分析：%prun和%run -p

代码的性能分析跟代码执行时间密切相关，只不过它关注的是耗费时间的位置。主要的Python性能分析工具是cProfile模块，它不是专为IPython设计的。cProfile在执行一个程序或代码块时，会记录各函数所耗费的时间。

cProfile一般是在命令行上使用的，它将执行整个程序然后输出各函数的执行时间。假设我们有一个简单的脚本：在一个循环中执行一些线性代数计算（计算一个100×100的矩阵的最大本征值绝对值）。

import numpy as npfrom numpy.linalg import eigvalsdef run*experiment(niter=100): K = 100 results = [] for* in xrange(niter): mat = np.random.randn(K, K) max_eigenvalue = np.abs(eigvals(mat)).max() results.append(max_eigenvalue) return resultssome_results = run_experiment()print 'Largest one we saw: %s' % np.max(some_results)

如果你还不懂NumPy，暂时先别管，后面会讲的。在命令行中输入下列命令即可通过cProfile启动该脚本：

python -m cProfile cprof_example.py

执行之后，你会发现输出结果是按函数名排序的。这让我们很难发现哪里才是最花时间的地方，因此通常都会再用-s标记指定一个排序规则：

$ python -m cProfile -s cumulative cprof*example.pyLargest one we saw: 11.923204422 15116 function calls (14927 primitive calls) in 0.720 secondsOrdered by: cumulative timencalls tottime percall cumtime percall filename:lineno(function)1 0.001 0.001 0.721 0.721 cprofexample.py:1(<module>)100 0.003 0.000 0.586 0.006 linalg.py:702(eigvals)200 0.572 0.003 0.572 0.003 {numpy.linalg.lapacklite.dgeev}1 0.002 0.002 0.075 0.075* ***init***.py:106(<module>)100 0.059 0.001 0.059 0.001 {method 'randn'}1 0.000 0.000 0.044 0.044 addnewdocs.py:9(<module>)2 0.001 0.001 0.037 0.019 ***init***.py:1(<module>)2 0.003 0.002 0.030 0.015 ***init***.py:2(<module>)1 0.000 0.000 0.030 0.030 typecheck.py:3(<module>)1 0.001 0.001 0.021 0.021 _init.py:15(<module>)1 0.013 0.013 0.013 0.013 numeric.py:1(<module>)1 0.000 0.000 0.009 0.009 __init.py:6(<module>)1 0.001 0.001 0.008 0.008 __init*.py:45(<module>)262 0.005 0.000 0.007 0.000 function_base.py:3178(add_newdoc)100 0.003 0.000 0.005 0.000 linalg.py:162(_assertFinite)...

这里只给出了输出结果中的前15行。只需查看cumtime列即可发现各函数所耗费的总时间。注意，如果一个函数调用了别的函数，计时器是不会停下来重新计时的。cProfile记录的是各函数调用的起始和结束时间，并依此计算总时间。

除命令行用法之外，cProfile还可以编程的方式分析任意代码块的性能。IPython为此提供了一个方便的接口，即%prun命令和带-p选项的%run。%prun的格式跟cProfile差不多，但它分析的是Python语句而不是整个.py文件：

In [4]: %prun -l 7 -s cumulative run_experiment() 4203 function calls in 0.643 secondsOrdered by: cumulative timeList reduced from 32 to 7 due to restriction <7>ncalls tottime percall cumtime percall filename:lineno(function) 1 0.000 0.000 0.643 0.643 <string>:1(<module>) 1 0.001 0.001 0.643 0.643 cprof_example.py:4(run_experiment) 100 0.003 0.000 0.583 0.006 linalg.py:702(eigvals) 200 0.569 0.003 0.569 0.003 {numpy.linalg.lapack_lite.dgeev} 100 0.058 0.001 0.058 0.001 {method 'randn'} 100 0.003 0.000 0.005 0.000 linalg.py:162(_assertFinite) 200 0.002 0.000 0.002 0.000 {method 'all' of 'numpy.ndarray' objects}

执行%run -p -s cumulative cprof_example.py也能达到上面那条系统命令行命令一样的效果，但是却无需退出IPython。

逐行分析函数性能

有些时候，从%prun（或其他基于cProfile的性能分析手段）得到的信息要么不足以说明函数的执行时间，要么就复杂到难以理解（按函数名聚合）。对于这种情况，我们可以使用一个叫做line_profiler的小型库（可以通过PyPI或随便一种包管理工具获取）。其中有一个新的魔术函数%lprun，它可以对一个或多个函数进行逐行性能分析。你可以修改IPython配置（参考IPython文件或本章稍后关于配置的内容）以启用这个扩展，代码如下所示：

A list of dotted module names of IPython extensions to load.c.TerminalIPythonApp.extensions = ['line_profiler']

line_profiler可以通过编程的方式使用（请参阅完整文档），但其最强大的一面却是在IPython中的交互式使用。假设你有一个prof_mod模块，其中有一些用于NumPy数组计算的代码，如下所示：

from numpy.random import randndef add_and_sum(x, y): added = x + y summed = added.sum(axis=1) return summeddef call_function(): x = randn(1000, 1000) y = randn(1000, 1000) return add_and_sum(x, y)

如果我们想了解add_and_sum函数的性能，%prun会给出如下所示的结果：

In [569]: %run prof_modIn [570]: x = randn(3000, 3000)In [571]: y = randn(3000, 3000)In [572]: %prun add_and_sum(x, y) 4 function calls in 0.049 secondsOrdered by: internal timencalls tottime percall cumtime percall filename:lineno(function) 1 0.036 0.036 0.046 0.046 prof_mod.py:3(add_and_sum) 1 0.009 0.009 0.009 0.009 {method 'sum' of 'numpy.ndarray' objects} 1 0.003 0.003 0.049 0.049 <string>:1(<module>) 1 0.000 0.000 0.000 0.000 {method 'disable' of '_lsprof.Profiler' objects}

这个结果并不能说明什么问题。启用line_profiler这个IPython扩展之后，就会出现一个新的魔术命令%lprun。用法上唯一的区别就是：必须为%lprun指明想要测试哪个或哪些函数。%lprun的通用语法为：

%lprun -f func1 -f func2 statement_to_profile

在本例中，我们想要测试的是add_and_sum，于是执行：

In [573]: %lprun -f add_and_sum add_and_sum(x, y)Timer unit: 1e-06 sFile: book_scripts/prof_mod.pyFunction: add_and_sum at line 3Total time: 0.045936 sLine # Hits Time Per Hit % Time Line Contents==
3 def add_and_sum(x, y): 4 1 36510 36510.0 79.5 added = x + y 5 1 9425 9425.0 20.5 summed = added.sum(axis=1) 6 1 1 1.0 0.0 return summed

这个结果就容易理解多了。这里我们测试的只是add_and_sum这一个函数。上面那个模块中还有一个call_function函数，我们可以结合add_and_sum一起测试，于是最终的测试命令就成了下面这个样子：

In [574]: %lprun -f add_and_sum -f call_function call_function()Timer unit: 1e-06 sFile: book_scripts/prof_mod.pyFunction: add_and_sum at line 3Total time: 0.005526 sLine # Hits Time Per Hit % Time Line Contents==
3 def add_and_sum(x, y): 4 1 4375 4375.0 79.2 added = x + y 5 1 1149 1149.0 20.8 summed = added.sum(axis=1) 6 1 2 2.0 0.0 return summedFile: book_scripts/prof_mod.pyFunction: call_function at line 8Total time: 0.121016 sLine # Hits Time Per Hit % Time Line Contents==
8 def call_function(): 9 1 57169 57169.0 47.2 x = randn(1000, 1000) 10 1 58304 58304.0 48.2 y = randn(1000, 1000) 11 1 5543 5543.0 4.6 return add_and_sum(x, y)

通常，我会用%prun（cProfile）做“宏观的”性能分析，而用%lprun（line_profiler）做“微观的”性能分析。这两个工具都很有必要了解一下。

注意： 在使用%lprun时，之所以必须显式指明待测试的函数名，是因为“跟踪”每一行代码的执行时间所需的开销很大。对不感兴趣的函数进行跟踪将会对性能分析结果造成显著的影响。

译注15

：第一，s不一定行，看提示，要用c；第二，这个s实际上是step into。

译注16

：也就是step over。

译注17

：作者在这里的意思是这种断点比较随便，是硬编码的。

IPython HTML Notebook

2011年，由Brian Granger领导的IPython团队开始开发一种基于Web技术的交互式计算文档格式，即IPython Notebook（见图3-4）。目前，它已经成为一种非常棒的交互式计算工具，同时还是科研和教学的一种理想媒介。本书中大部分示例都是用它编写的.我强烈建议你也试试。

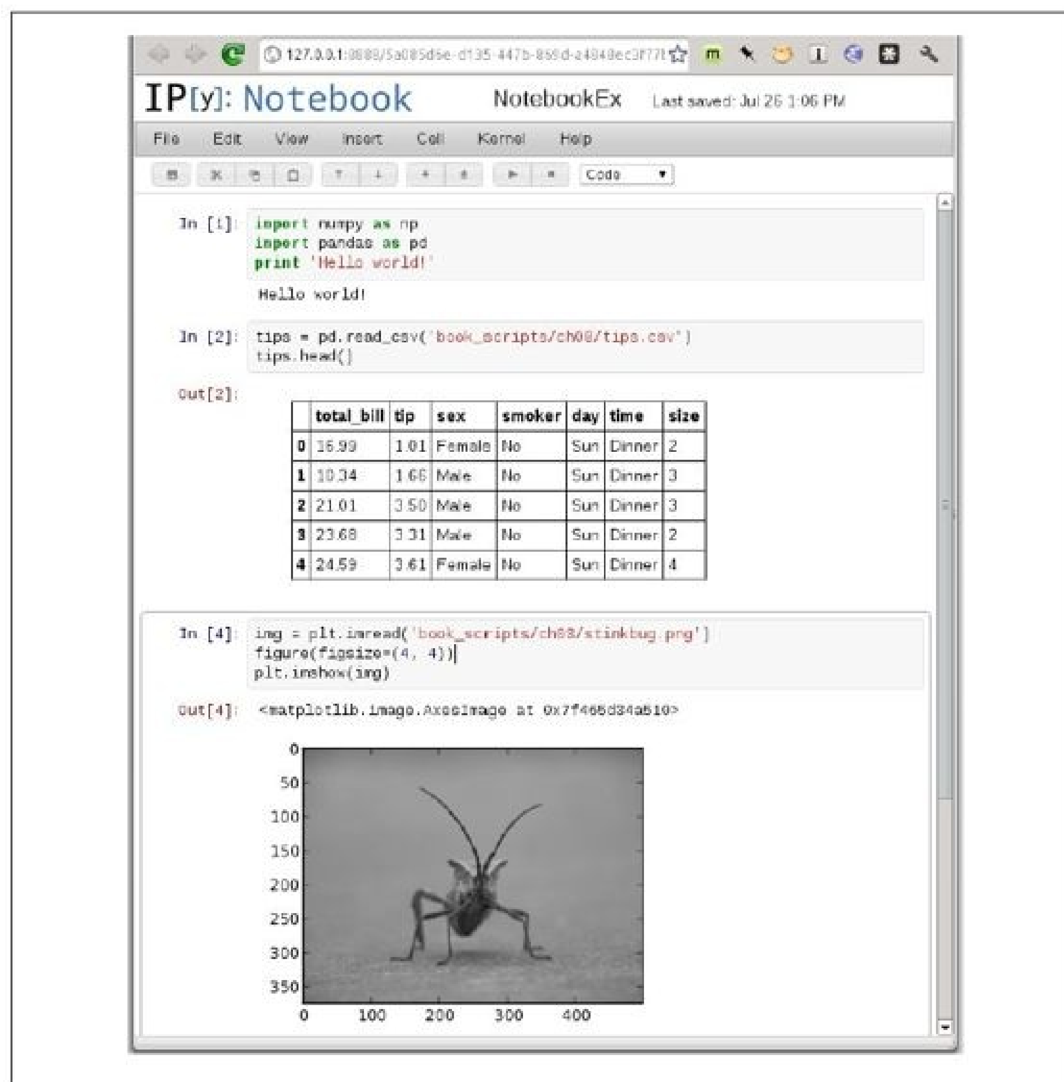


图3-4：IPython Notebook

它有一种基于JSON的文档格式.ipynb，使你可以轻松分享代码、输出结果以及图片等内容。目前在各种Python研讨会上，一种流行的演示手段就是使用IPython Notebook，然后再将.ipynb文件发布到网上以供所有人查阅。

IPython Notebook应用程序是一个运行于命令行上的轻量级服务器进程。执行下面这条命令即可启动：

$ ipython notebook --pylab=inline[NotebookApp] Using existing profile dir: u'/home/wesm/.config/ipython/profile_default' [NotebookApp] Serving notebooks from /home/wesm/book_scripts[NotebookApp] The IPython Notebook is running at: http://127.0.0.1:8888/[NotebookApp] Use Control-C to stop this server and shut down all kernels.

在大多数平台上，你的首选Web浏览器会自动打开Notebook的仪表板（dashboard）。有时你可能需要手工打开上面列出的那个URL。你可以在这里创建一个新的记事本并开始研究工作。

由于我们是在一个Web浏览器中使用Notebook的，因此该服务器进程可以运行于任何地方。你甚至可以连接到那些运行在云服务（如Amazon EC2）上的Notebook。直到写作本书时为止，一个新的名为NotebookCloud（http://notebookcloud.appspot.com）的项目已经诞生了，它可以轻松地在Amazon EC2上启动记事本。

利用IPython提高代码开发效率的几点提示

为了在IPython中开发、调试代码，并充分发挥其交互优势，许多用户都需要转换一下工作模式。像编码风格以及一些操作细节可能需要做一些调整。

就这点来说，本节的内容更像是艺术而非科学，你需要有一些编程经验才好判断其能否提高你的工作效率。总之，你得让你的代码结构更易于交互且结果更易于查看。我发现通过IPython设计的软件要比独立的命令行应用程序好用。当你执行自己或别人在几个月甚至几年前编写的代码时出现了错误，想找出问题所在时，IPython的交互性就会变得非常重要。

重新加载模块依赖项

在Python中，当你输入import some_lib时，some_lib中的代码就会被执行，且其中所有的变量、函数和引入项都会被保存在一个新建的some_lib模块命名空间中。下次你再输入import some_lib时，就会得到这个模块命名空间的一个引用。而这对于IPython的交互式代码开发模式就会有一个问题，比如说，用%run执行的某段脚本中牵扯到了某个刚刚做了修改的模块。假设我们有一个test_script.py文件，其中有下列代码：

import some_libx = 5y = [1, 2, 3, 4]result = some_lib.get_answer(x, y)

如果在执行了%run test*script.py之后又对somelib.py进行了修改，下次再执行%run testscript.py时将仍然会使用老版的somelib。其原因就是Python的“一次加载”模块系统。这个行为不同于其他一些数据分析环境（如MATLAB，它会自动应用代码修改[注1](#809468440711498-Zhu_1_You_Yu_Yi_Ge_Mo_Kuai_Huo_Bao_Ke_Neng_Hui_Zai_Yi_Ge_Cheng_Xu_Zhong_De_Bu_Tong_Wei_Zhi_Duo_Ci_Yin_Ru__Suo_Yi_PythonHui_Zai_Di_Yi_Ci_Yin_Ru_Zhe_Xie_Mo_Kuai_Shi)*）。为了解决这个问题，你有两个办法可用。第一个办法是使用Python内置的reload函数。将test_script.py修改成下面这个样子：

import some_libreload(some_lib)x = 5y = [1, 2, 3, 4]result = some_lib.get_answer(x, y)

这样就保证每次执行test_script.py时都能用上最新版的some_lib了。显然，当依赖变得更强时，就需要在很多地方插入很多的reload。对于这个问题，IPython提供了一个特殊的dreload函数（非魔术函数）来解决模块的“深度”（递归）重加载。如果执行import some_lib之后再输入dreload(some_lib)，则它会尝试重新加载some_lib及其所有的依赖项。遗憾的是，这个办法也不是万灵丹，但是如果真的不行了，重启IPython就行了。

代码设计提示

这个问题不太好讲，但我在日常工作中确实发现了一些高层次的原则。

保留有意义的对象和数据

人们一般不会在命令行上编写下面这样的程序：

from my*functions import gdef f(x, y): return g(x + y)def main(): x = 6 y = 7.5 result = x + yif* ***name*** == '*_main*': main()

如果我们在IPython中执行这段代码的话会出现什么问题？我们在IPython shell中将访问不到任何结果以及main函数中定义的对象。好点的办法是直接在该模块的全局命名空间中执行main中的代码（如果你希望该模块是可引入的，也可以将这些代码放在if **name**=='**main**':块中）。这样，当你%run这段代码时，就能看到main中定义的所有变量了。对这个简单的例子而言，这个原则意义不大，但对本书后面将要介绍的那些针对大数据集的复杂数据分析问题而言就很重要了。

扁平结构要比嵌套结构好

深度嵌套的代码让我想到了洋葱。在测试或调试函数时，你要把这个洋葱剥多少层才能找到感兴趣的代码？“扁平结构要比嵌套结构好”的思想来自"Zen of Python"译注18，它对交互式的代码开发模式同样有效。编写函数和类时应尽量注意低耦合和模块化，这样可以使它们更易于测试（如果你编写单元测试的话）、调试和交互式使用。

无惧大文件

如果曾经学过Java（或其他类似的语言），可能会有人告诉你要“尽量保持文件的小型化”。在许多语言中，这都是一个不错的建议。长度太长通常是一种不好的“臭代码”，意味着需要重构或重组。然而在IPython中开发代码时，处理10个小的（但互相关联的）文件（比如都低于100行）可能会让你更为头疼，还不如直接一个大文件或两三个大点的文件来得痛快。更少的文件意味着需要重新加载的模块更少，编辑时需要在各个文件之间的跳转次数也更少。我发现维护更大的（具有高内聚度的）模块会更实用也更具有Python特点。在解决完问题之后，有时将大文件拆分成小文件会更好。

显然，我并不建议将此原则极端化，那可能会让你将所有代码都放到一个巨大的文件里面。对一个大型代码库而言，要找到一种合乎逻辑的模块/包架构需要花点工夫，但这对团队工作非常重要。每个模块都应该具有足够高的内聚度，而且要能足够直观地找到对应各种功能的函数和类。

注1

：由于一个模块或包可能会在一个程序中的不同位置多次引入，所以Python会在第一次引入这些模块时对其进行缓存，而不是每次都执行模块中的代码。否则，应用程序的模块化和良好的代码组织等手段就达不到高效的目的了。

译注18

：这是Tim Peters 2004年写的一首“诗”，执行"import this"就能看到。有网民将其翻译成三字经的形式（又名“蛇宗三字经”）。另外，有兴趣的话，可以看看this的源代码。

高级IPython功能

让你的类对IPython更加友好

IPython力求为各种对象呈现一个友好的字符串表示。对于许多对象（如字典、列表和元组等），内置的pprint模块就能给出漂亮的格式。但是对于你自己定义的那些类，就必须自己生成所需的字符串输出。假设我们有下面这个简单的类：

class Message: def **init**(self, msg): self.msg = msg

如果像下面这样写，你就会失望地发现这个类的默认输出形式非常不好看：

In [576]: x = Message('I have a secret')In [577]: xOut[577]: <__main__.Message instance="" at="" 0x60ebbd8="">

由于IPython会获取**repr**方法返回的字符串（具体办法是output=repr(obj)），并将其显示到控制台上。因此，我们可以为上面那个类添加一个简单的**repr**方法以得到一个更有意义的输出形式：

class Message: def **init**(self, msg): self.msg = msg def **repr**(self): return 'Message: %s' % self.msgIn [579]: x = Message('I have a secret')In [580]: xOut[580]: Message: I have a secret

个性化和配置

IPython shell在外观（如颜色、提示符、行间距等）和行为方面的大部分内容都是可以进行配置的。下面是能够通过配置做的部分事情：

·修改颜色方案。

·修改输入输出提示符。

·去掉Out提示符跟下一个In提示符之间的空行。

·执行任意Python语句。这些语句可以用于引入所有常用的东西，还可以做一些你希望每次启动IPython都发生的事情。

·启用IPython扩展，如line_profiler中的魔术命令%lprun。

·定义你自己的魔术命令或系统别名。

所有这些配置选项都定义在一个叫做ipython_config.py的文件中，可以在~/.config/ipython/目录（UNIX）和%HOME%/.ipython/目录（Windows）中找到。具体的主目录取决于你的系统。配置信息是基于特定个性化设置的。一般来说，正常启动IPython将会加载默认的个性化设置（位于profile_default目录中）。因此，在我的Linux系统中，默认IPython配置文件的完整路径是：

/home/wesm/.config/ipython/profile_default/ipython_config.py

这里我就不对该文件的内容作详细介绍了。因为其注释已经说明了各个配置项的功能，各位读者完全可以自己照着做。还有一个很实用的功能是拥有多个个性化设置。假设你想要专门为某个应用程序或项目量身定做一套IPython配置。输入下面这样的命令即可新建一个个性化设置：

ipython profile create secret_project

然后编辑新建的这个profile_secret_project目录中的配置文件，再用下面这种方式启动IPython：

$ ipython --profile=secret_projectPython 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul 3 2011, 15:17:51)Type "copyright", "credits" or "license" for more information.IPython 0.13 -- An enhanced Interactive Python.? -> Introduction and overview of IPython's features.%quickref -> Quick reference.help -> Python's own help system.object? -> Details about 'object', use 'object??' for extra details.IPython profile: secret_projectIn [1]:

同样，有关个性化和配置方面的详细信息，请参考IPython的在线文档。

致谢

本章的部分内容由IPython Development Team整理。我对他们创建了如此神奇的工具而感激涕零。

第4章　NumPy基础：数组和矢量计算

NumPy（Numerical Python的简称）是高性能科学计算和数据分析的基础包。它是本书所介绍的几乎所有高级工具的构建基础。其部分功能如下：

·ndarray，一个具有矢量算术运算和复杂广播能力的快速且节省空间的多维数组。

·用于对整组数据进行快速运算的标准数学函数（无需编写循环）。

·用于读写磁盘数据的工具以及用于操作内存映射文件的工具。

·线性代数、随机数生成以及傅里叶变换功能。

·用于集成由C、C++、Fortran等语言编写的代码的工具。

最后一点也是从生态系统角度来看最重要的一点。由于NumPy提供了一个简单易用的C API，因此很容易将数据传递给由低级语言编写的外部库，外部库也能以NumPy数组的形式将数据返回给Python。这个功能使Python成为一种包装C/C++/Fortran历史代码库的选择，并使被包装库拥有一个动态的、易用的接口。

NumPy本身并没有提供多么高级的数据分析功能，理解NumPy数组以及面向数组的计算将有助于你更加高效地使用诸如pandas之类的工具。如果你是Python新手，而且只是想用pandas随便处理一下数据就行，那就跳过本章吧，没关系的。更多NumPy高级功能（比如广播），请参见第12章。

对于大部分数据分析应用而言，我最关注的功能主要集中在：

·用于数据整理和清理、子集构造和过滤、转换等快速的矢量化数组运算。

·常用的数组算法，如排序、唯一化、集合运算等。

·高效的描述统计和数据聚合/摘要运算。

·用于异构数据集的合并/连接运算的数据对齐和关系型数据运算。

·将条件逻辑表述为数组表达式（而不是带有if-elif-else分支的循环）。

·数据的分组运算（聚合、转换、函数应用等）。第5章将对此进行详细讲解。

虽然NumPy提供了这些功能的计算基础，但你可能还是想将pandas作为数据分析工作的基础（尤其是对于结构化或表格化数据），因为它提供了能使大部分常见数据任务变得非常简洁的丰富高级接口。pandas还提供了一些NumPy所没有的更加领域特定的功能，如时间序列处理等。

注意： 在本章以及本书中，我将依照标准的NumPy约定，即总是使用import numpy as np。当然，你也可以为了不写np.而直接在代码中使用from numpy import *，但我得提醒你最好还是不要养成这样的坏习惯。

NumPy的ndarray：一种多维数组对象

NumPy最重要的一个特点就是其N维数组对象（即ndarray），该对象是一个快速而灵活的大数据集容器。你可以利用这种数组对整块数据执行一些数学运算，其语法跟标量元素之间的运算一样：

In [8]: dataOut[8]:array([[0.9526, -0.246 , -0.8856], [0.5639, 0.2379, 0.9104]])In [9]: data * 10 In [10]: data + dataOut[9]: Out[10]:array([[9.5256, -2.4601, -8.8565], array([[1.9051, -0.492 , -1.7713], [5.6385, 2.3794, 9.104]]) [1.1277, 0.4759, 1.8208]])

ndarray是一个通用的同构数据多维容器，也就是说，其中的所有元素必须是相同类型的。每个数组都有一个shape（一个表示各维度大小的元组）和一个dtype（一个用于说明数组数据类型的对象）：

In [11]: data.shapeOut[11]: (2, 3)In [12]: data.dtypeOut[12]: dtype('float64')

本章将会介绍NumPy数组的基本用法，这对于本书后面各章的理解基本够用。虽然大多数数据分析工作不需要深入理解NumPy，但是精通面向数组的编程和思维方式是成为Python科学计算牛人的一大关键步骤。

注意： 当你在本书中看到“数组”、“NumPy数组”、"ndarray"时，基本上都指的是同一样东西，即ndarray对象。

创建ndarray

创建数组最简单的办法就是使用array函数。它接受一切序列型的对象（包括其他数组），然后产生一个新的含有传入数据的NumPy数组。以一个列表的转换为例：

In [13]: data1 = [6, 7.5, 8, 0, 1]In [14]: arr1 = np.array(data1)In [15]: arr1Out[15]: array([6. , 7.5, 8. , 0. , 1.])

嵌套序列（比如由一组等长列表组成的列表）将会被转换为一个多维数组：

In [16]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]In [17]: arr2 = np.array(data2)In [18]: arr2Out[18]:array([[1, 2, 3, 4], [5, 6, 7, 8]])In [19]: arr2.ndimOut[19]: 2In [20]: arr2.shapeOut[20]: (2, 4)

除非显式说明（稍后将会详细介绍），np.array会尝试为新建的这个数组推断出一个较为合适的数据类型。数据类型保存在一个特殊的dtype对象中。比如说，在上面的两个例子中，我们有：

In [21]: arr1.dtypeOut[21]: dtype('float64')In [22]: arr2.dtypeOut[22]: dtype('int64')

除np.array之外，还有一些函数也可以新建数组。比如，zeros和ones分别可以创建指定长度或形状的全0或全1数组。empty可以创建一个没有任何具体值的数组。要用这些方法创建多维数组，只需传入一个表示形状的元组即可：

In [23]: np.zeros(10)Out[23]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])In [24]: np.zeros((3, 6))Out[24]:array([[0., 0., 0., 0., 0., 0.], [0., 0., 0., 0., 0., 0.], [0., 0., 0., 0., 0., 0.]])In [25]: np.empty((2, 3, 2))Out[25]:array([[[4.94065646e-324, 4.94065646e-324], [3.87491056e-297, 2.46845796e-130], [4.94065646e-324, 4.94065646e-324]], [[1.90723115e+083, 5.73293533e-053], [-2.33568637e+124, -6.70608105e-012], [4.42786966e+160, 1.27100354e+025]]])

警告： 认为np.empty会返回全0数组的想法是不安全的。很多情况下（如前所示），它返回的都是一些未初始化的垃圾值。

arange是Python内置函数range的数组版：

In [26]: np.arange(15)Out[26]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])

表4-1列出了一些数组创建函数。由于NumPy关注的是数值计算，因此，如果没有特别指定，数据类型基本都是float64（浮点数）。

表4-1：数组创建函数

函数	说明
array	将输入数据（列表、元组、数组或其他序列类型）转换为ndarray。要么推断出dtype，要么显式指定dtype。默认直接复制输入数据
asarray	将输入转换为ndarray，如果输入本身就是一个ndarray就不进行复制
arange	类似于内置的range，但返回的是一个ndarray而不是列表
ones、ones_like	根据指定的形状和dtype创建一个全1数组。ones_like以另一个数组为参数，并根据其形状和dtype创建一个全1数组
zeros、zeros_like	类似于ones和ones_like，只不过产生的是全0数组而已
empty、empty_like	创建新数组，只分配内存空间但不填充任何值
eye、identity	创建一个正方的N×N单位矩阵（对角线为1，其余为0）

ndarray的数据类型

dtype（数据类型）是一个特殊的对象，它含有ndarray将一块内存解释为特定数据类型所需的信息：

In [27]: arr1 = np.array([1, 2, 3], dtype=np.float64)In [28]: arr2 = np.array([1, 2, 3], dtype=np.int32)In [29]: arr1.dtype In [30]: arr2.dtypeOut[29]: dtype('float64') Out[30]: dtype('int32')

dtype是NumPy如此强大和灵活的原因之一。多数情况下，它们直接映射到相应的机器表示，这使得“读写磁盘上的二进制数据流”以及“集成低级语言代码（如C、Fortran）”等工作变得更加简单。数值型dtype的命名方式相同：一个类型名（如float或int），后面跟一个用于表示各元素位长的数字。标准的双精度浮点值（即Python中的float对象）需要占用8字节（即64位）。因此，该类型在NumPy中就记作float64。表4-2列出了NumPy所支持的全部数据类型。

注意：记不住这些NumPy的dtype也没关系，新手更是如此。通常只需要知道你所处理的数据的大致类型是浮点数、复数、整数、布尔值、字符串，还是普通的Python对象即可。当你需要控制数据在内存和磁盘中的存储方式时（尤其是对大数据集），那就得了解如何控制存储类型。

表4-2：NumPy的数据类型

类型	类型代码	说明
int8、uint8	i1、u1	有符号和无符号的8位（1个字节）整型
int16、uint16	i2、u2	有符号和无符号的16位（2个字节）整型
int32、uint32	i4、u4	有符号和无符号的32位（4个字节）整型
int64、uint64	i8、u8	有符号和无符号的64位（8个字节）整型
float16	f2	半精度浮点数
float32	f4或f	标准的单精度浮点数。与C的float兼容
float64	f8或d	标准的双精度浮点数。与C的double和Python的float对象兼容
float128	f16或g	扩展精度浮点数
complex64、complex128、complex256	c8、c16、c32	分别用两个32位、64位或128位浮点数表示的复数
bool	?	存储True和False值的布尔类型

表4-2：NumPy的数据类型（续）

类型	类型代码	说明
object	O	Python对象类型
string_	S	固定长度的字符串类型（每个字符1个字节）。例如，要创建一个长度为10的字符串，应使用S10
unicode_	U	固定长度的unicode类型（字节数由平台决定）。跟字符串的定义方式一样（如U10）

你可以通过ndarray的astype方法显式地转换其dtype：

In [31]: arr = np.array([1, 2, 3, 4, 5])In [32]: arr.dtypeOut[32]: dtype('int64')In [33]: float_arr = arr.astype(np.float64)In [34]: float_arr.dtypeOut[34]: dtype('float64')

在本例中，整数被转换成了浮点数。如果将浮点数转换成整数，则小数部分将会被截断：

In [35]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])In [36]: arrOut[36]: array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])In [37]: arr.astype(np.int32)Out[37]: array([3, -1, -2, 0, 12, 10], dtype=int32)

如果某字符串数组表示的全是数字，也可以用astype将其转换为数值形式：

In [38]: numeric*strings = np.array(['1.25', '-9.6', '42'], dtype=np.string*)In [39]: numeric_strings.astype(float)Out[39]: array([1.25, -9.6 , 42.])

如果转换过程因为某种原因而失败了（比如某个不能被转换为float64的字符串），就会引发一个TypeError。看到了吧，我比较懒，写的是float而不是np.float64；NumPy很聪明，它会将Python类型映射到等价的dtype上。

数组的dtype还有另外一个用法：

In [40]: int_array = np.arange(10)In [41]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)In [42]: int_array.astype(calibers.dtype)Out[42]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])

你还可以用简洁的类型代码来表示dtype：

In [43]: empty_uint32 = np.empty(8, dtype='u4')In [44]: empty_uint32Out[44]:array([0, 0, 65904672, 0, 64856792, 0, 39438163, 0], dtype=uint32)

注意：调用astype无论如何都会创建出一个新的数组（原始数据的一份拷贝），即使新dtype跟老dtype相同也是如此。

警告：注意，浮点数（比如float64和float32）只能表示近似的分数值。在复杂计算中，由于可能会积累一些浮点错误，因此比较操作只能在一定小数位以内有效。

数组和标量之间的运算

数组很重要，因为它使你不用编写循环即可对数据执行批量运算。这通常就叫做矢量化（vectorization）。大小相等的数组之间的任何算术运算都会将运算应用到元素级：

In [45]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])In [46]: arrOut[46]:array([[1., 2., 3.], [4., 5., 6.]])In [47]: arr * arr In [48]: arr - arrOut[47]: Out[48]:array([[1., 4., 9.], array([[0., 0., 0.], [16., 25., 36.]]) [0., 0., 0.]])

同样，数组与标量的算术运算也会将那个标量值传播到各个元素：

In [49]: 1 / arr In [50]: arr ** 0.5Out[49]: Out[50]:array([[1. , 0.5 , 0.3333], array([[1. , 1.4142, 1.7321], [0.25 , 0.2 , 0.1667]]) [2. , 2.2361, 2.4495]])

不同大小的数组之间的运算叫做广播（broadcasting），我们将在第12章中对其进行详细讨论。本书的内容不需要对广播机制有多深的理解。

基本的索引和切片

NumPy数组的索引是一个内容丰富的主题，因为选取数据子集或单个元素的方式有很多。一维数组很简单。从表面上看，它们跟Python列表的功能差不多：

In [51]: arr = np.arange(10)In [52]: arrOut[52]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])In [53]: arr[5]Out[53]: 5In [54]: arr[5:8]Out[54]: array([5, 6, 7])In [55]: arr[5:8] = 12In [56]: arrOut[56]: array([0, 1, 2, 3, 4, 12, 12, 12, 8, 9])

如上所示，当你将一个标量值赋值给一个切片时（如arr[5:8]=12），该值会自动传播（也就说后面将会讲到的“广播”）到整个选区。跟列表最重要的区别在于，数组切片是原始数组的视图。这意味着数据不会被复制，视图上的任何修改都会直接反映到源数组上：

In [57]: arr_slice = arr[5:8]In [58]: arr_slice[1] = 12345In [59]: arrOut[59]: array([0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])In [60]: arr_slice[:] = 64In [61]: arrOut[61]: array([0, 1, 2, 3, 4, 64, 64, 64, 8, 9])

如果你刚开始接触NumPy，可能会对此感到惊讶（尤其是当你曾经用过其他热衷于复制数组数据的编程语言）。由于NumPy的设计目的是处理大数据，所以你可以想象一下，假如NumPy坚持要将数据复制来复制去的话会产生何等的性能和内存问题。

警告：如果你想要得到的是ndarray切片的一份副本而非视图，就需要显式地进行复制操作，例如arr[5:8].copy()。

对于高维度数组，能做的事情更多。在一个二维数组中，各索引位置上的元素不再是标量而是一维数组：

In [62]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])In [63]: arr2d[2]Out[63]: array([7, 8, 9])

因此，可以对各个元素进行递归访问，但这样需要做的事情有点多。你可以传入一个以逗号隔开的索引列表来选取单个元素。也就是说，下面两种方式是等价的：

In [64]: arr2d[0][2]Out[64]: 3In [65]: arr2d[0, 2]Out[65]: 3

图4-1说明了二维数组的索引方式。

图4-1：NumPy数组中的元素索引

在多维数组中，如果省略了后面的索引，则返回对象会是一个维度低一点的ndarray（它含有高一级维度上的所有数据译注1）。因此，在2×2×3数组arr3d中：

In [66]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])In [67]: arr3dOut[67]:array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

arr3d[0]是一个2×3数组：

In [68]: arr3d[0]Out[68]:array([[1, 2, 3], [4, 5, 6]])

标量值和数组都可以被赋值给arr3d[0]：

In [69]: old_values = arr3d[0].copy()In [70]: arr3d[0] = 42In [71]: arr3dOut[71]:array([[[42, 42, 42], [42, 42, 42]], [[7, 8, 9], [10, 11, 12]]])In [72]: arr3d[0] = old_valuesIn [73]: arr3dOut[73]:array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

以此类推，arr3d[1,0]可以访问索引以(1,0)开头的那些值（以一维数组的形式返回）：

In [74]: arr3d[1, 0]Out[74]: array([7, 8, 9])

注意，在上面所有这些选取数组子集的例子中，返回的数组都是视图。

切片索引

ndarray的切片语法跟Python列表这样的一维对象差不多：

In [75]: arr[1:6]Out[75]: array([1, 2, 3, 4, 64])

高维度对象的花样更多，你可以在一个或多个轴上进行切片，也可以跟整数索引混合使用。对于上面那个二维数组arr2d，其切片方式稍显不同：

In [76]: arr2d In [77]: arr2d[:2]Out[76]: Out[77]:array([[1, 2, 3], array([[1, 2, 3], [4, 5, 6], [4, 5, 6]]) [7, 8, 9]])

可以看出，它是沿着第0轴（即第一个轴）切片的。也就是说，切片是沿着一个轴向选取元素的。你可以一次传入多个切片，就像传入多个索引那样：

In [78]: arr2d[:2, 1:]Out[78]:array([[2, 3], [5, 6]])

像这样进行切片时，只能得到相同维数的数组视图。通过将整数索引和切片混合，可以得到低维度的切片：

In [79]: arr2d[1, :2] In [80]: arr2d[2, :1]Out[79]: array([4, 5]) Out[80]: array([7])

图4-2对此进行了说明。注意，“只有冒号”表示选取整个轴，因此你可以像下面这样只对高维轴进行切片：

In [81]: arr2d[:, :1]Out[81]:array([[1], [4], [7]])

自然，对切片表达式的赋值操作也会被扩散到整个选区：

In [82]: arr2d[:2, 1:] = 0

布尔型索引

来看这样一个例子，假设我们有一个用于存储数据的数组以及一个存储姓名的数组（含有重复项）。在这里，我将使用numpy.random中的randn函数生成一些正态分布的随机数据：

In [83]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])In [84]: data = randn(7, 4)In [85]: namesOut[85]:array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='|S4')In [86]: dataOut[86]:array([[-0.048 , 0.5433, -0.2349, 1.2792], [-0.268 , 0.5465, 0.0939, -2.0445],

[-0.047 , -2.026 , 0.7719, 0.3103], [2.1452, 0.8799, -0.0523, 0.0672], [-1.0023, -0.1698, 1.1503, 1.7289], [0.1913, 0.4544, 0.4519, 0.5535], [0.5994, 0.8174, -0.9297, -1.2564]])

Expression	Shape
arr[:2, 1:]	(2, 2)
arr[2]	(3,)
arr[2, :]	(3,)
arr[2:, :]	(1, 3)
arr[:, :2]	(3, 2)
arr[1, :2]	(2,)
arr[1:2, :2]	(1, 2)

图4-2：二维数组切片

假设每个名字都对应data数组中的一行，而我们想要选出对应于名字"Bob"的所有行。跟算术运算一样，数组的比较运算（如==）也是矢量化的。因此，对names和字符串"Bob"的比较运算将会产生一个布尔型数组：

In [87]: names == 'Bob'Out[87]: array([True, False, False, True, False, False, False], dtype=bool)

这个布尔型数组可用于数组索引：

In [88]: data[names == 'Bob']Out[88]:array([[-0.048 , 0.5433, -0.2349, 1.2792], [2.1452, 0.8799, -0.0523, 0.0672]])

布尔型数组的长度必须跟被索引的轴长度一致。此外，还可以将布尔型数组跟切片、整数（或整数序列，稍后将对此进行详细讲解）混合使用：

In [89]: data[names == 'Bob', 2:]Out[89]:array([[-0.2349, 1.2792], [-0.0523, 0.0672]])In [90]: data[names == 'Bob', 3]Out[90]: array([1.2792, 0.0672])

要选择除"Bob"以外的其他值，既可以使用不等于符号（!=），也可以通过负号（-）对条件进行否定：

In [91]: names != 'Bob'Out[91]: array([False, True, True, False, True, True, True], dtype=bool)In [92]: data[-(names == 'Bob')]Out[92]:array([[-0.268 , 0.5465, 0.0939, -2.0445], [-0.047 , -2.026 , 0.7719, 0.3103], [-1.0023, -0.1698, 1.1503, 1.7289], [0.1913, 0.4544, 0.4519, 0.5535], [0.5994, 0.8174, -0.9297, -1.2564]])

选取这三个名字中的两个需要组合应用多个布尔条件，使用&（和）、|（或）之类的布尔算术运算符即可：

In [93]: mask = (names == 'Bob') | (names == 'Will')In [94]: maskOut[94]: array([True, False, True, True, True, False, False], dtype=bool)In [95]: data[mask]Out[95]:array([[-0.048 , 0.5433, -0.2349, 1.2792], [-0.047 , -2.026 , 0.7719, 0.3103], [2.1452, 0.8799, -0.0523, 0.0672], [-1.0023, -0.1698, 1.1503, 1.7289]])

通过布尔型索引选取数组中的数据，将总是创建数据的副本，即使返回一模一样的数组也是如此。

警告： Python关键字and和or在布尔型数组中无效。

通过布尔型数组设置值是一种经常用到的手段。为了将data中的所有负值都设置为0，我们只需：

In [96]: data[data < 0] = 0In [97]: dataOut[97]:array([[0. , 0.5433, 0. , 1.2792], [0. , 0.5465, 0.0939, 0.], [0. , 0. , 0.7719, 0.3103], [2.1452, 0.8799, 0. , 0.0672], [0. , 0. , 1.1503, 1.7289], [0.1913, 0.4544, 0.4519, 0.5535], [0.5994, 0.8174, 0. , 0.]])

通过一维布尔数组设置整行或列的值也很简单：

In [98]: data[names != 'Joe'] = 7In [99]: dataOut[99]:array([[7. , 7. , 7. , 7.], [0. , 0.5465, 0.0939, 0.], [7. , 7. , 7. , 7.], [7. , 7. , 7. , 7.], [7. , 7. , 7. , 7.], [0.1913, 0.4544, 0.4519, 0.5535], [0.5994, 0.8174, 0. , 0.]])

花式索引

花式索引（Fancy indexing）是一个NumPy术语，它指的是利用整数数组进行索引。假设我们有一个8×4数组：

In [100]: arr = np.empty((8, 4))In [101]: for i in range(8): ....: arr[i] = iIn [102]: arrOut[102]:array([[0., 0., 0., 0.], [1., 1., 1., 1.], [2., 2., 2., 2.], [3., 3., 3., 3.], [4., 4., 4., 4.], [5., 5., 5., 5.], [6., 6., 6., 6.], [7., 7., 7., 7.]])

为了以特定顺序选取行子集，只需传入一个用于指定顺序的整数列表或ndarray即可：

In [103]: arr[[4, 3, 0, 6]]Out[103]:array([[4., 4., 4., 4.], [3., 3., 3., 3.], [0., 0., 0., 0.], [6., 6., 6., 6.]])

这段代码确实达到我们的要求了！使用负数索引将会从末尾开始选取行：

In [104]: arr[[-3, -5, -7]]Out[104]:array([[5., 5., 5., 5.], [3., 3., 3., 3.], [1., 1., 1., 1.]])

一次传入多个索引数组会有一点特别。它返回的是一个一维数组，其中的元素对应各个索引元组：

有关reshape的知识将在第12章中讲解In [105]: arr = np.arange(32).reshape((8, 4))In [106]: arrOut[106]:array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15], [16, 17, 18, 19], [20, 21, 22, 23], [24, 25, 26, 27], [28, 29, 30, 31]])In [107]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]Out[107]: array([4, 23, 29, 10])

我们来看看具体是怎么一回事。最终选出的是元素(1,0)、(5,3)、(7,1)和(2,2)。这个花式索引的行为可能会跟某些用户的预期不一样（包括我在内），选取矩阵的行列子集应该是矩形区域的形式才对。下面是得到该结果的一个办法：

In [108]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]Out[108]:array([[4, 7, 5, 6], [20, 23, 21, 22], [28, 31, 29, 30], [8, 11, 9, 10]])

另外一个办法是使用np.ix_函数，它可以将两个一维整数数组转换为一个用于选取方形区域的索引器：

In [109]: arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]Out[109]:array([[4, 7, 5, 6], [20, 23, 21, 22], [28, 31, 29, 30], [8, 11, 9, 10]])

记住，花式索引跟切片不一样，它总是将数据复制到新数组中。

数组转置和轴对换

转置（transpose）是重塑的一种特殊形式，它返回的是源数据的视图（不会进行任何复制操作）。数组不仅有transpose方法，还有一个特殊的T属性：

In [110]: arr = np.arange(15).reshape((3, 5))In [111]: arrOut[111]:array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14]])In [112]: arr.TOut[112]:array([[0, 5, 10], [1, 6, 11], [2, 7, 12], [3, 8, 13], [4, 9, 14]])

在进行矩阵计算时，经常需要用到该操作，比如利用np.dot计算矩阵内积XTX：

In [113]: arr = np.random.randn(6, 3)In [114]: np.dot(arr.T, arr)Out[114]:array([[2.584 , 1.8753, 0.8888], [1.8753, 6.6636, 0.3884], [0.8888, 0.3884, 3.9781]])

对于高维数组，transpose需要得到一个由轴编号组成的元组才能对这些轴进行转置（比较费脑子）：

In [115]: arr = np.arange(16).reshape((2, 2, 4))In [116]: arrOut[116]:array([[[0, 1, 2, 3], [4, 5, 6, 7]], [[8, 9, 10, 11], [12, 13, 14, 15]]])In [117]: arr.transpose((1, 0, 2))Out[117]:array([[[0, 1, 2, 3], [8, 9, 10, 11]], [[4, 5, 6, 7], [12, 13, 14, 15]]])

简单的转置可以使用.T，它其实就是进行轴对换而已。ndarray还有一个swapaxes方法，它需要接受一对轴编号：

In [118]: arrOut[118]:array([[[0, 1, 2, 3], [4, 5, 6, 7]], [[8, 9, 10, 11], [12, 13, 14, 15]]])In [119]: arr.swapaxes(1, 2)Out[119]:array([[[0, 4], [1, 5], [2, 6], [3, 7]], [[8, 12], [9, 13], [10, 14], [11, 15]]])

swapaxes也是返回源数据的视图（不会进行任何复制操作）。

译注1

：括号外面的“维度”是一维、二维、三维、四维之类的意思，而括号里面的应该理解为“轴”。也就是说，这里指的是“返回的低维数组含有原始高维数组某条轴上的所有数据”。

通用函数：快速的元素级数组函数

通用函数（即ufunc）是一种对ndarray中的数据执行元素级运算的函数。你可以将其看做简单函数（接受一个或多个标量值，并产生一个或多个标量值）的矢量化包装器。

许多ufunc都是简单的元素级变体，如sqrt和exp：

In [120]: arr = np.arange(10)In [121]: np.sqrt(arr)Out[121]:array([0. , 1. , 1.4142, 1.7321, 2. , 2.2361, 2.4495, 2.6458, 2.8284, 3.])In [122]: np.exp(arr)Out[122]:array([1. , 2.7183, 7.3891, 20.0855, 54.5982, 148.4132, 403.4288, 1096.6332, 2980.958 , 8103.0839])

这些都是一元（unary）ufunc。另外一些（如add或maximum）接受2个数组（因此也叫二元（binary）ufunc），并返回一个结果数组：

In [123]: x = randn(8)In [124]: y = randn(8)In [125]: xOut[125]:array([0.0749, 0.0974, 0.2002, -0.2551, 0.4655, 0.9222, 0.446 , -0.9337])In [126]: yOut[126]:array([0.267 , -1.1131, -0.3361, 0.6117, -1.2323, 0.4788, 0.4315, -0.7147])In [127]: np.maximum(x, y) # 元素级最大值Out[127]:array([0.267 , 0.0974, 0.2002, 0.6117, 0.4655, 0.9222, 0.446 , -0.7147])

虽然并不常见，但有些ufunc的确可以返回多个数组。modf就是一个例子，它是Python内置函数divmod的矢量化版本，用于浮点数数组的小数和整数部分。

In [128]: arr = randn(7) * 5In [129]: np.modf(arr)Out[129]:(array([-0.6808, 0.0636, -0.386 , 0.1393, -0.8806, 0.9363, -0.883]), array([-2., 4., -3., 5., -3., 3., -6.]))

表4-3和表4-4分别列出了一些一元和二元ufunc。

表4-3：一元ufunc

函数	说明
abs、fabs	计算整数、浮点数或复数的绝对值。对于非复数值，可以使用更快的fabs
sqrt	计算各元素的平方根。相当于arr ** 0.5
square	计算各元素的平方。相当于arr ** 2
exp	计算各元素的指数e^x
log、log10、log2、log1p	分别为自然对数（底数为e）、底数为10的log、底数为2的log、log(1 + x)
sign	计算各元素的正负号：1（正数）、0（零）、－1（负数）
ceil	计算各元素的ceiling值，即大于等于该值的最小整数
floor	计算各元素的floor值，即小于等于该值的最大整数
rint	将各元素值四舍五入到最接近的整数，保留dtype
modf	将数组的小数和整数部分以两个独立数组的形式返回
isnan	返回一个表示“哪些值是NaN（这不是一个数字）”的布尔型数组
isfinite、isinf	分别返回一个表示“哪些元素是有穷的（非inf，非NaN）”或“哪些元素是无穷的”的布尔型数组
cos、cosh、sin、sinh、tan、tanh	普通型和双曲型三角函数

表4-3：一元ufunc（续）

函数	说明
arccos、arccosh、arcsin、arcsinh、arctan、arctanh	反三角函数
logical_not	计算各元素not x的真值。相当于-arr

表4-4：二元ufunc

函数	说明
add	将数组中对应的元素相加
subtract	从第一个数组中减去第二个数组中的元素
multiply	数组元素相乘
divide、floor_divide	除法或向下圆整除法（丢弃余数）
power	对第一个数组中的元素A，根据第二个数组中的相应元素B，计算A^B
maximum、fmax	元素级的最大值计算。fmax将忽略NaN
minimum、fmin	元素级的最小值计算。fmin将忽略NaN
mod	元素级的求模计算（除法的余数）
copysign	将第二个数组中的值的符号复制给第一个数组中的值

greater、greater_equal、less、less_equal、equal、not_equal	执行元素级的比较运算，最终产生布尔型数组。相当于中缀运算符>、>=、<、<=、==、!=
logical_and、logical_or、logical_xor	执行元素级的真值逻辑运算。相当于中缀运算符&、|、^

利用数组进行数据处理

NumPy数组使你可以将许多种数据处理任务表述为简洁的数组表达式（否则需要编写循环）。用数组表达式代替循环的做法，通常被称为矢量化。一般来说，矢量化数组运算要比等价的纯Python方式快上一两个数量级（甚至更多），尤其是各种数值计算。在后面内容中（见第12章）我将介绍广播，这是一种针对矢量化计算的强大手段。

假设我们想要在一组值（网格型）上计算函数sqrt(x^2+y^2)。np.meshgrid函数接受两个一维数组，并产生两个二维矩阵（对应于两个数组中所有的(x,y)对）：

In [130]: points = np.arange(-5, 5, 0.01) # 1000个间隔相等的点In [131]: xs, ys = np.meshgrid(points, points)In [132]: ysOut[132]:array([[-5. , -5. , -5. , ..., -5. , -5. , -5.], [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99], [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98], ..., [4.97, 4.97, 4.97, ..., 4.97, 4.97, 4.97], [4.98, 4.98, 4.98, ..., 4.98, 4.98, 4.98], [4.99, 4.99, 4.99, ..., 4.99, 4.99, 4.99]])

现在，对该函数的求值运算就好办了，把这两个数组当做两个浮点数那样编写表达式即可：

In [134]: import matplotlib.pyplot as pltIn [135]: z = np.sqrt(xs **2 + ys** 2)In [136]: zOut[136]:array([[7.0711, 7.064 , 7.0569, ..., 7.0499, 7.0569, 7.064], [7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569], [7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499], ..., [7.0499, 7.0428, 7.0357, ..., 7.0286, 7.0357, 7.0428], [7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499], [7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569]])In [137]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()Out[137]: In [138]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")Out[138]:

函数值（一个二维数组）的图形化结果如图4-3所示。这张图我是用matplotlib的imshow函数创建的。

将条件逻辑表述为数组运算

numpy.where函数是三元表达式x if condition else y的矢量化版本。假设我们有一个布尔数组和两个值数组：

In [140]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])In [141]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])In [142]: cond = np.array([True, False, True, True, False])

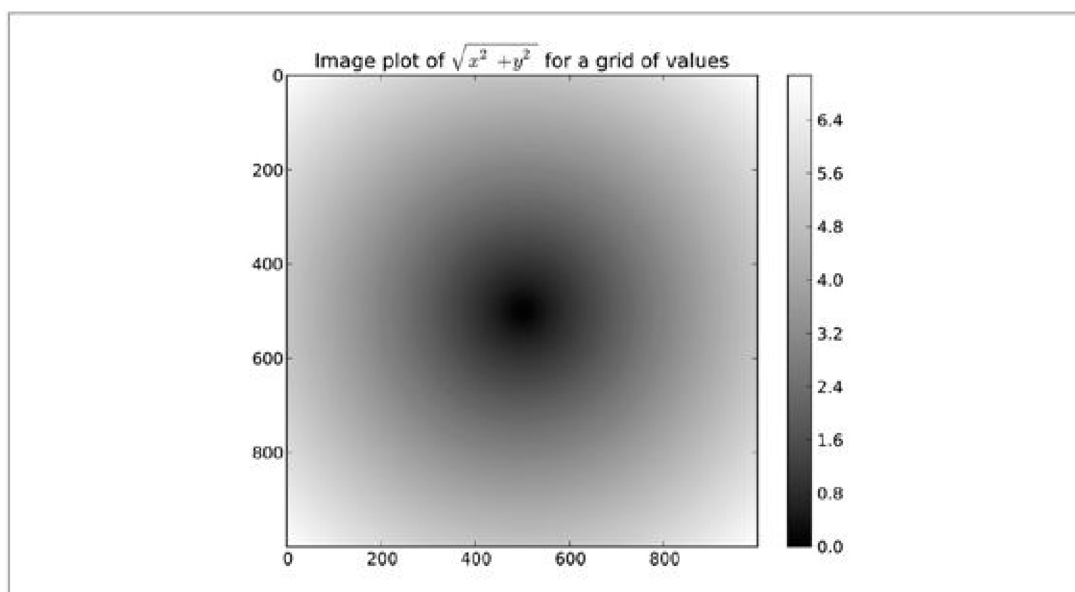


图4-3：根据网格对函数求值的结果

假设我们想要根据cond中的值选取xarr和yarr的值：当cond中的值为True时，选取xarr的值，否则从yarr中选取。列表推导式的写法应该如下所示：

In [143]: result = [(x if c else y)   ...: for x, y, c in zip(xarr, yarr, cond)]In [144]: resultOut[144]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]

这有几个问题。第一，它对大数组的处理速度不是很快（因为所有工作都是由纯Python完成的）。第二，无法用于多维数组。若使用np.where，则可以将该功能写得非常简洁：

In [145]: result = np.where(cond, xarr, yarr)In [146]: resultOut[146]: array([1.1, 2.2, 1.3, 1.4, 2.5])

np.where的第二个和第三个参数不必是数组，它们都可以是标量值。在数据分析工作中，where通常用于根据另一个数组而产生一个新的数组。假设有一个由随机数据组成的矩阵，你希望将所有正值替换为2，将所有负值替换为－2。若利用np.where，则会非常简单：

In [147]: arr = randn(4, 4)In [148]: arrOut[148]:array([[0.6372, 2.2043, 1.7904, 0.0752], [-1.5926, -1.1536, 0.4413, 0.3483], [-0.1798, 0.3299, 0.7827, -0.7585], [0.5857, 0.1619, 1.3583, -1.3865]])In [149]: np.where(arr > 0, 2, -2)Out[149]:array([[2, 2, 2, 2], [-2, -2, 2, 2], [-2, 2, 2, -2], [2, 2, 2, -2]])In [150]: np.where(arr > 0, 2, arr) # 只将正值设置为2Out[150]:array([[2. , 2. , 2. , 2.], [-1.5926, -1.1536, 2. , 2.], [-0.1798, 2. , 2. , -0.7585], [2. , 2. , 2. , -1.3865]])

传递给where的数组大小可以不相等，甚至可以是标量值。

只要稍微动动脑子，你就能用where表述出更复杂的逻辑。想象一下这样一个例子，我有两个布尔型数组cond1和cond2，希望根据4种不同的布尔值组合实现不同的赋值操作：

result = []for i in range(n): if cond1[i] and cond2[i]: result.append(0) elif cond1[i]: result.append(1) elif cond2[i]: result.append(2) else: result.append(3)

虽然不是非常明显，但这个for循环确实可以被改写成一个嵌套的where表达式：

np.where(cond1 & cond2, 0, np.where(cond1, 1, np.where(cond2, 2, 3)))

在这个特殊的例子中，我们还可以利用“布尔值在计算过程中可以被当做0或1处理”这个事实，所以还能将其写成下面这样的算术运算（虽然看上去有点神秘）：

result = 1 *(cond1 -cond2)* + 2 (cond2 & -cond1) + 3 * -(cond1 | cond2)

数学和统计方法

可以通过数组上的一组数学函数对整个数组或某个轴向的数据进行统计计算。sum、mean以及标准差std等聚合计算（aggregation，通常叫做约简（reduction））既可以当做数组的实例方法调用，也可以当做顶级NumPy函数使用：

In [151]: arr = np.random.randn(5, 4) # 正态分布的数据In [152]: arr.mean()Out[152]: 0.062814911084854597In [153]: np.mean(arr)Out[153]: 0.062814911084854597In [154]: arr.sum()Out[154]: 1.2562982216970919

mean和sum这类的函数可以接受一个axis参数（用于计算该轴向上的统计值），最终结果是一个少一维的数组：

In [155]: arr.mean(axis=1)Out[155]: array([-1.2833, 0.2844, 0.6574, 0.6743, -0.0187])In [156]: arr.sum(0)Out[156]: array([-3.1003, -1.6189, 1.4044, 4.5712])

其他如cumsum和cumprod之类的方法则不聚合，而是产生一个由中间结果组成的数组：

In [157]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])In [158]: arr.cumsum(0)Out[158]:array([[0, 1, 2], [3, 5, 7], [9, 12, 15]])In [159]: arr.cumprod(1)Out[159]:array([[0, 0, 0], [3, 12, 60], [6, 42, 336]])

表4-5列出了全部的基本数组统计方法。后续章节中有很多例子都会用到这些方法。

表4-5：基本数组统计方法

方法	说明
sum	对数组中全部或某轴向的元素求和。零长度的数组的sum为0
mean	算术平均数。零长度的数组的mean为NaN
std、var	分别为标准差和方差，自由度可调（默认为n）
min、max	最大值和最小值
argmin、argmax	分别为最大和最小元素的索引

表4-5：基本数组统计方法（续）

方法	说明
cumsum	所有元素的累计和
cumprod	所有元素的累计积

用于布尔型数组的方法

在上面这些方法中，布尔值会被强制转换为1（True）和0（False）。因此，sum经常被用来对布尔型数组中的True值计数：

In [160]: arr = randn(100)In [161]: (arr > 0).sum() # 正值的数量Out[161]: 44

另外还有两个方法any和all，它们对布尔型数组非常有用。any用于测试数组中是否存在一个或多个True，而all则检查数组中所有值是否都是True：

In [162]: bools = np.array([False, False, True, False])In [163]: bools.any()Out[163]: TrueIn [164]: bools.all()Out[164]: False

这两个方法也能用于非布尔型数组，所有非0元素将会被当做True。

排序

跟Python内置的列表类型一样，NumPy数组也可以通过sort方法就地排序：

In [165]: arr = randn(8)In [166]: arrOut[166]:array([0.6903, 0.4678, 0.0968, -0.1349, 0.9879, 0.0185, -1.3147, -0.5425])In [167]: arr.sort()In [168]: arrOut[168]:array([-1.3147, -0.5425, -0.1349, 0.0185, 0.0968, 0.4678, 0.6903, 0.9879])

多维数组可以在任何一个轴向上进行排序，只需将轴编号传给sort即可：

In [169]: arr = randn(5, 3)In [170]: arrOut[170]:array([[-0.7139, -1.6331, -0.4959], [0.8236, -1.3132, -0.1935], [-1.6748, 3.0336, -0.863], [-0.3161, 0.5362, -2.468], [0.9058, 1.1184, -1.0516]])In [171]: arr.sort(1)In [172]: arrOut[172]:array([[-1.6331, -0.7139, -0.4959], [-1.3132, -0.1935, 0.8236], [-1.6748, -0.863 , 3.0336], [-2.468 , -0.3161, 0.5362], [-1.0516, 0.9058, 1.1184]])

顶级方法np.sort返回的是数组的已排序副本，而就地排序则会修改数组本身。计算数组分位数最简单的办法是对其进行排序，然后选取特定位置的值：

In [173]: large_arr = randn(1000)In [174]: large_arr.sort()In [175]: large_arr[int(0.05 * len(large_arr))] # 5%分位数Out[175]: -1.5791023260896004

更多关于NumPy排序方法以及诸如间接排序之类的高级技术，请参阅第12章。在pandas中还可以找到一些其他跟排序有关的数据操作（比如根据一列或多列对表格型数据进行排序）。

唯一化以及其他的集合逻辑

NumPy提供了一些针对一维ndarray的基本集合运算。最常用的可能要数np.unique了，它用于找出数组中的唯一值并返回已排序的结果：

In [176]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])In [177]: np.unique(names)Out[177]:array(['Bob', 'Joe', 'Will'], dtype='|S4')In [178]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])In [179]: np.unique(ints)Out[179]: array([1, 2, 3, 4])

拿跟np.unique等价的纯Python代码来对比一下：

In [180]: sorted(set(names))Out[180]: ['Bob', 'Joe', 'Will']

另一个函数np.in1d用于测试一个数组中的值在另一个数组中的成员资格，返回一个布尔型数组：

In [181]: values = np.array([6, 0, 0, 3, 2, 5, 6])In [182]: np.in1d(values, [2, 3, 6])Out[182]: array([True, False, False, True, True, False, True], dtype=bool)

NumPy中的集合函数请参见表4-6。

表4-6：数组的集合运算

方法	说明
unique(x)	计算x中的唯一元素，并返回有序结果
intersect1d(x, y)	计算x和y中的公共元素，并返回有序结果
union1d(x, y)	计算x和y的并集，并返回有序结果
in1d(x, y)	得到一个表示“x的元素是否包含于y”的布尔型数组
setdiff1d(x, y)	集合的差，即元素在x中且不在y中
setxor1d(x, y)	集合的对称差，即存在于一个数组中但不同时存在于两个数组中的元素[译注2]

译注2：简单点说，就是“异或”。

用于数组的文件输入输出

NumPy能够读写磁盘上的文本数据或二进制数据。后面的章节将会告诉你一些pandas中用于将表格型数据读取到内存的工具。

将数组以二进制格式保存到磁盘

np.save和np.load是读写磁盘数组数据的两个主要函数。默认情况下，数组是以未压缩的原始二进制格式保存在扩展名为.npy的文件中的。

In [183]: arr = np.arange(10)In [184]: np.save('some_array', arr)

如果文件路径末尾没有扩展名.npy，则该扩展名会被自动加上。然后就可以通过np.load读取磁盘上的数组：

In [185]: np.load('some_array.npy')Out[185]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

通过np.savez可以将多个数组保存到一个压缩文件中，将数组以关键字参数的形式传入即可：

In [186]: np.savez('array_archive.npz', a=arr, b=arr)

加载.npz文件时，你会得到一个类似字典的对象，该对象会对各个数组进行延迟加载：

In [187]: arch = np.load('array_archive.npz')In [188]: arch['b']Out[188]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

存取文本文件

从文件中加载文本是一个非常标准的任务。Python中的文件读写函数的格式很容易将新手搞晕，所以我将主要介绍pandas中的read_csv和read_table函数。有时，我们需要用np.loadtxt或更为专门化的np.genfromtxt将数据加载到普通的NumPy数组中。

这些函数都有许多选项可供使用：指定各种分隔符、针对特定列的转换器函数、需要跳过的行数等。以一个简单的逗号分隔文件（CSV）为例：

In [191]: !cat array_ex.txt译注30.580052,0.186730,1.040717,1.1344110.194163,-0.636917,-0.938659,0.124094-0.126410,0.268607,-0.695724,0.047428-1.484413,0.004176,-0.744203,0.0054872.302869,0.200131,1.670238,-1.881090-0.193230,1.047233,0.482803,0.960334

该文件可以被加载到一个二维数组中，如下所示：

In [192]: arr = np.loadtxt('array_ex.txt', delimiter=',')In [193]: arrOut[193]:array([[0.5801, 0.1867, 1.0407, 1.1344], [0.1942, -0.6369, -0.9387, 0.1241], [-0.1264, 0.2686, -0.6957, 0.0474], [-1.4844, 0.0042, -0.7442, 0.0055], [2.3029, 0.2001, 1.6702, -1.8811], [-0.1932, 1.0472, 0.4828, 0.9603]])

np.savetxt执行的是相反的操作：将数组写到以某种分隔符隔开的文本文件中。genfromtxt跟loadtxt差不多，只不过它面向的是结构化数组和缺失数据处理。更多有关结构化数组的知识，请参阅第12章。

注意： 更多有关文件读写（尤其是表格型数据）的知识，请参阅本书后面有关pandas和DataFrame对象的章节。

译注3

：这是Linux的，Windows得用type。

线性代数

线性代数（如矩阵乘法、矩阵分解、行列式以及其他方阵数学等）是任何数组库的重要组成部分。不像某些语言（如MATLAB），通过*对两个二维数组相乘得到的是一个元素级的积，而不是一个矩阵点积。因此，NumPy提供了一个用于矩阵乘法的dot函数（既是一个数组方法也是numpy命名空间中的一个函数）：

In [194]: x = np.array([[1., 2., 3.], [4., 5., 6.]])In [195]: y = np.array([[6., 23.], [-1, 7], [8, 9]])In [196]: x In [197]: yOut[196]: Out[197]:array([[1., 2., 3.], array([[6., 23.], [4., 5., 6.]]) [-1., 7.], [8., 9.]])In [198]: x.dot(y) # 相当于np.dot(x, y)Out[198]:array([[28., 64.], [67., 181.]])

一个二维数组跟一个大小合适的一维数组的矩阵点积运算之后将会得到一个一维数组：

In [199]: np.dot(x, np.ones(3))Out[199]: array([6., 15.])

numpy.linalg中有一组标准的矩阵分解运算以及诸如求逆和行列式之类的东西。它们跟MATLAB和R等语言所使用的是相同的行业标准级Fortran库，如BLAS、LAPACK、Intel MKL（可能有，取决于你的NumPy版本）等：

In [201]: from numpy.linalg import inv, qrIn [202]: X = randn(5, 5)In [203]: mat = X.T.dot(X)In [204]: inv(mat)Out[204]:array([[3.0361, -0.1808, -0.6878, -2.8285, -1.1911], [-0.1808, 0.5035, 0.1215, 0.6702, 0.0956], [-0.6878, 0.1215, 0.2904, 0.8081, 0.3049], [-2.8285, 0.6702, 0.8081, 3.4152, 1.1557], [-1.1911, 0.0956, 0.3049, 1.1557, 0.6051]])In [205]: mat.dot(inv(mat))Out[205]:array([[1., 0., 0., 0., -0.], [0., 1., -0., 0., 0.], [0., -0., 1., 0., 0.], [0., -0., -0., 1., -0.], [0., 0., 0., 0., 1.]])In [206]: q, r = qr(mat)In [207]: rOut[207]:array([[-6.9271, 7.389 , 6.1227, -7.1163, -4.9215], [0. , -3.9735, -0.8671, 2.9747, -5.7402], [0. , 0. , -10.2681, 1.8909, 1.6079], [0. , 0. , 0. , -1.2996, 3.3577], [0. , 0. , 0. , 0. , 0.5571]])

表4-7中列出了一些最常用的线性代数函数。

注意： Python科学计算社区盼望着有朝一日能实现矩阵乘法的中缀运算符，以便能用一种更漂亮的语法代替np.dot。不过目前就只能先这样了。

表4-7：常用的numpy.linalg函数

函数	说明
diag	以一维数组的形式返回方阵的对角线（或非对角线）元素，或将一维数组转换为方阵（非对角线元素为0）
dot	矩阵乘法
trace	计算对角线元素的和
det	计算矩阵行列式
eig	计算方阵的本征值和本征向量
inv	计算方阵的逆
pinv	计算矩阵的Moore-Penrose伪逆
qr	计算QR分解
svd	计算奇异值分解（SVD）
solve	解线性方程组Ax = b，其中A为一个方阵
lstsq	计算Ax = b的最小二乘解

随机数生成

numpy.random模块对Python内置的random进行了补充，增加了一些用于高效生成多种概率分布的样本值的函数。例如，你可以用normal来得到一个标准正态分布的4×4样本数组：

In [208]: samples = np.random.normal(size=(4, 4))In [209]: samplesOut[209]:array([[0.1241, 0.3026, 0.5238, 0.0009], [1.3438, -0.7135, -0.8312, -2.3702], [-1.8608, -0.8608, 0.5601, -1.2659], [0.1198, -1.0635, 0.3329, -2.3594]])

而Python内置的random模块则只能一次生成一个样本值。从下面的测试结果中可以看出，如果需要产生大量样本值，numpy.random快了不止一个数量级：

In [210]: from random import normalvariateIn [211]: N = 1000000In [212]: %timeit samples = [normalvariate(0, 1) for _ in xrange(N)]1 loops, best of 3: 1.33 s per loopIn [213]: %timeit np.random.normal(size=N)10 loops, best of 3: 57.7 ms per loop

表4-8列出了numpy.random中的部分函数。在下一节中，我将给出一些利用这些函数一次性生成大量样本值的范例。

表4-8：部分numpy.random函数

函数	说明
seed	确定随机数生成器的种子
permutation	返回一个序列的随机排列或返回一个随机排列的范围
shuffle	对一个序列就地随机排列
rand	产生均匀分布的样本值
randint	从给定的上下限范围内随机选取整数
randn	产生正态分布（平均值为0，标准差为1）的样本值，类似于MATLAB接口
binomial	产生二项分布的样本值
normal	产生正态（高斯）分布的样本值
beta	产生Beta分布的样本值
chisquare	产生卡方分布的样本值
gamma	产生Gamma分布的样本值
uniform	产生在[0, 1)中均匀分布的样本值

范例：随机漫步

我们通过模拟随机漫步来说明如何运用数组运算。先来看一个简单的随机漫步的例子：从0开始，步长1和－1出现的概率相等。我们通过内置的random模块以纯Python的方式实现1000步的随机漫步：

import randomposition = 0walk = [position]steps = 1000for i in xrange(steps): step = 1 if random.randint(0, 1) else -1 position += step walk.append(position)

图4-4是根据前100个随机漫步值生成的折线图。

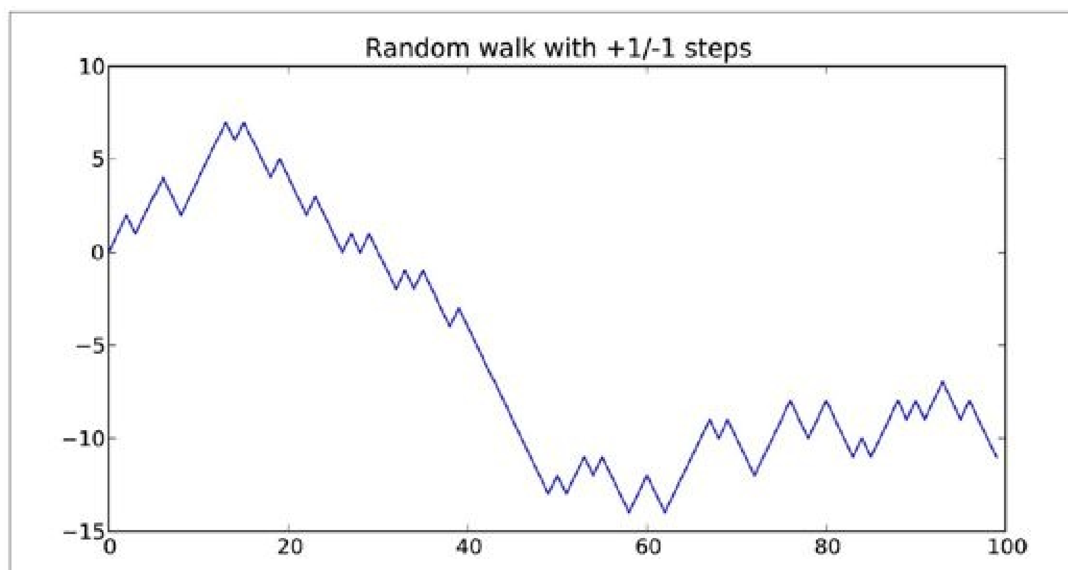


图4-4：简单的随机漫步

不难看出，这其实就是随机漫步中各步的累计和，可以用一个数组运算来实现。因此，我用np.random模块一次性随机产生1000个“掷硬币”结果（即两个数中任选一个），将其分别设置为1或－1，然后计算累计和：

In [215]: nsteps = 1000In [216]: draws = np.random.randint(0, 2, size=nsteps)In [217]: steps = np.where(draws > 0, 1, -1)In [218]: walk = steps.cumsum()

有了这些数据之后，我们就可以做一些统计工作了，比如求取最大值和最小值：

In [219]: walk.min()Out[219]: -3In [220]: walk.max()Out[220]: 31

现在来看一个复杂点的统计任务——首次穿越时间，即随机漫步过程中第一次到达某个特定值的时间。假设我们想要知道本次随机漫步需要多久才能距离初始0点至少10步远（任一方向均可）。np.abs(walk)>=10可以得到一个布尔型数组，它表示的是距离是否达到或超过10，而我们想要知道的是第一个10或－10的索引。可以用argmax来解决这个问题，它返回的是该布尔型数组第一个最大值的索引（True就是最大值）：

In [221]: (np.abs(walk) >= 10).argmax()Out[221]: 37

注意，这里使用argmax并不是很高效，因为它无论如何都会对数组进行完全扫描。在本例中，只要发现了一个True，那我们就知道它是个最大值了。

一次模拟多个随机漫步

如果你希望模拟多个随机漫步过程（比如5000个），只需对上面的代码做一点点修改即可生成所有的随机漫步过程。只要给numpy.random的函数传入一个二元元组就可以产生一个二维数组，然后我们就可以一次性计算5000个随机漫步过程（一行一个）的累计和了：

In [222]: nwalks = 5000In [223]: nsteps = 1000In [224]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0或1In [225]: steps = np.where(draws > 0, 1, -1)In [226]: walks = steps.cumsum(1)In [227]: walksOut[227]:array([[1, 0, 1, ..., 8, 7, 8], [1, 0, -1, ..., 34, 33, 32], [1, 0, -1, ..., 4, 5, 4], ..., [1, 2, 1, ..., 24, 25, 26], [1, 2, 3, ..., 14, 13, 14], [-1, -2, -3, ..., -24, -23, -22]])

现在，我们来计算所有随机漫步过程的最大值和最小值：

In [228]: walks.max()Out[228]: 138In [229]: walks.min()Out[229]: -133

得到这些数据之后，我们来计算30或－30的最小穿越时间。这里得要稍微动一下脑筋，因为不是5000个过程都到达了30。我们可以用any方法来对此进行检查：

In [230]: hits30 = (np.abs(walks) >= 30).any(1)In [231]: hits30Out[231]: array([False, True, False, ..., False, True, False], dtype=bool)In [232]: hits30.sum() # 到达30或－30的数量Out[232]: 3410

然后我们利用这个布尔型数组选出那些穿越了30（绝对值）的随机漫步（行），并调用argmax在轴1上获取穿越时间：

In [233]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)In [234]: crossing_times.mean()Out[234]: 498.88973607038122

请尝试用其他分布方式得到漫步数据。只需使用不同的随机数生成函数即可，如normal用于生成指定均值和标准差的正态分布数据：

In [235]: steps = np.random.normal(loc=0, scale=0.25, ...: size=(nwalks, nsteps))

第5章　pandas入门

pandas是本书后续内容的首选库。它含有使数据分析工作变得更快更简单的高级数据结构和操作工具。pandas是基于NumPy构建的，让以NumPy为中心的应用变得更加简单。

先介绍一点背景。我是在2008年早期还在AQR（一家定量投资管理公司）任职期间开始着手构建pandas的。那时候，没有任何一个单独的工具能够满足我工作上的全部需求：

·具备按轴自动或显式数据对齐功能的数据结构。这可以防止许多由于数据未对齐以及来自不同数据源（索引方式不同）的数据而导致的常见错误。

·集成时间序列功能。

·既能处理时间序列数据也能处理非时间序列数据的数据结构。

·数学运算和约简（比如对某个轴求和）可以根据不同的元数据（轴编号）执行。

·灵活处理缺失数据。

·合并及其他出现在常见数据库（例如基于SQL的）中的关系型运算。

我希望能够在一个地方完成所有这些事情，最好是一种能进行通用软件开发的语言。Python是一门不错的候选语言，但那时候它还没有一组能完全提供上述功能的数据结构和工具。

在过去的4年中，pandas逐渐成长为一个非常大的库，它所能解决的数据处理问题已经比我期望的要多得多了。但随着其范围的扩大，它也逐渐背离了我最初所期望的简洁性和易用性。我希望你在读完本书之后，也能像我一样认为它是一个不可或缺的工具。

在本书后续部分中，我将使用下面这样的pandas引入约定：

In [1]: from pandas import Series, DataFrameIn [2]: import pandas as pd

因此，只要你在代码中看到pd.，就得想到这是pandas。因为Series和DataFrame用的次数非常多，所以将其引入本地命名空间中会更方便。

pandas的数据结构介绍

要使用pandas，你首先就得熟悉它的两个主要数据结构：Series和DataFrame。虽然它们并不能解决所有问题，但它们为大多数应用提供了一种可靠的、易于使用的基础。

Series

Series是一种类似于一维数组的对象，它由一组数据（各种NumPy数据类型）以及一组与之相关的数据标签（即索引）组成。仅由一组数据即可产生最简单的Series：

In [4]: obj = Series([4, 7, -5, 3])In [5]: objOut[5]:0 41 72 -53 3

Series的字符串表现形式为：索引在左边，值在右边。由于我们没有为数据指定索引，于是会自动创建一个0到N1（N为数据的长度）的整数型索引。你可以通过Series 的values和index属性获取其数组表示形式和索引对象：

In [6]: obj.valuesOut[6]: array([4, 7, -5, 3])In [7]: obj.indexOut[7]: Int64Index([0, 1, 2, 3])

通常，我们希望所创建的Series带有一个可以对各个数据点进行标记的索引：

In [8]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])In [9]: obj2Out[9]:d 4b 7a -5c 3In [10]: obj2.indexOut[10]: Index([d, b, a, c], dtype=object)

与普通NumPy数组相比，你可以通过索引的方式选取Series中的单个或一组值：

In [11]: obj2['a']Out[11]: -5In [12]: obj2['d'] = 6In [13]: obj2[['c', 'a', 'd']]Out[13]:c 3a -5d 6

NumPy数组运算（如根据布尔型数组进行过滤、标量乘法、应用数学函数等）都会保留索引和值之间的链接：

In [14]: obj2Out[14]:d 6b 7a -5c 3In [15]: obj2[obj2 > 0] In [16]: obj2 * 2 In [17]: np.exp(obj2)Out[15]: Out[16]: Out[17]:d 6 d 12 d 403.428793b 7 b 14 b 1096.633158c 3 a -10 a 0.006738 c 6 c 20.085537

还可以将Series看成是一个定长的有序字典，因为它是索引值到数据值的一个映射。它可以用在许多原本需要字典参数的函数中：

In [18]: 'b' in obj2Out[18]: TrueIn [19]: 'e' in obj2Out[19]: False

如果数据被存放在一个Python字典中，也可以直接通过这个字典来创建Series：

In [20]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}In [21]: obj3 = Series(sdata)In [22]: obj3Out[22]:Ohio 35000Oregon 16000Texas 71000Utah 5000

如果只传入一个字典，则结果Series中的索引就是原字典的键（有序排列）。

In [23]: states = ['California', 'Ohio', 'Oregon', 'Texas']In [24]: obj4 = Series(sdata, index=states)In [25]: obj4Out[25]:California NaNOhio 35000Oregon 16000Texas 71000

在这个例子中，sdata中跟states索引相匹配的那3个值会被找出来并放到相应的位置上，但由于"California"所对应的sdata值找不到，所以其结果就为NaN（即“非数字”（not a number），在pandas中，它用于表示缺失或NA值）。我将使用缺失（missing）或NA表示缺失数据。pandas的isnull和notnull函数可用于检测缺失数据：

In [26]: pd.isnull(obj4) In [27]: pd.notnull(obj4)Out[26]: Out[27]:California True California FalseOhio False Ohio TrueOregon False Oregon TrueTexas False Texas True

Series也有类似的实例方法：

In [28]: obj4.isnull()Out[28]:California TrueOhio FalseOregon FalseTexas False

我将在本章详细讲解如何处理缺失数据。

对于许多应用而言，Series最重要的一个功能是：它在算术运算中会自动对齐不同索引的数据。

In [29]: obj3 In [30]: obj4Out[29]: Out[30]:Ohio 35000 California NaNOregon 16000 Ohio 35000Texas 71000 Oregon 16000Utah 5000 Texas 71000In [31]: obj3 + obj4Out[31]:California NaNOhio 70000Oregon 32000Texas 142000Utah NaN

数据对齐功能将在一个单独的主题中讲解。

Series对象本身及其索引都有一个name属性，该属性跟pandas其他的关键功能关系非常密切：

In [32]: obj4.name = 'population'In [33]: obj4.index.name = 'state'In [34]: obj4Out[34]:stateCalifornia NaNOhio 35000Oregon 16000Texas 71000Name: population

Series的索引可以通过赋值的方式就地修改：

In [35]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']In [36]: objOut[36]:Bob 4Steve 7Jeff -5Ryan 3

DataFrame

DataFrame是一个表格型的数据结构，它含有一组有序的列，每列可以是不同的值类型（数值、字符串、布尔值等）。DataFrame既有行索引也有列索引，它可以被看做由Series组成的字典（共用同一个索引）。跟其他类似的数据结构相比（如R的data.frame），DataFrame中面向行和面向列的操作基本上是平衡的。其实，DataFrame中的数据是以一个或多个二维块存放的（而不是列表、字典或别的一维数据结构）。有关DataFrame内部的技术细节远远超出了本书所讨论的范围。

注意：虽然DataFrame是以二维结构保存数据的，但你仍然可以轻松地将其表示为更高维度的数据（层次化索引的表格型结构，这是pandas中许多高级数据处理功能的关键要素，我们稍后再来讨论这个问题）。

构建DataFrame的办法有很多，最常用的一种是直接传入一个由等长列表或NumPy数组组成的字典：

data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'], 'year': [2000, 2001, 2002, 2001, 2002], 'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}frame = DataFrame(data)

结果DataFrame会自动加上索引（跟Series一样），且全部列会被有序排列：

In [38]: frameOut[38]: pop state year0 1.5 Ohio 20001 1.7 Ohio 20012 3.6 Ohio 20023 2.4 Nevada 20014 2.9 Nevada 2002

如果指定了列序列，则DataFrame的列就会按照指定顺序进行排列：

In [39]: DataFrame(data, columns=['year', 'state', 'pop'])Out[39]: year state pop0 2000 Ohio 1.51 2001 Ohio 1.72 2002 Ohio 3.63 2001 Nevada 2.44 2002 Nevada 2.9

跟Series一样，如果传入的列在数据中找不到，就会产生NA值：

In [40]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'], ...: index=['one', 'two', 'three', 'four', 'five'])In [41]: frame2Out[41]: year state pop debtone 2000 Ohio 1.5 NaNtwo 2001 Ohio 1.7 NaNthree 2002 Ohio 3.6 NaNfour 2001 Nevada 2.4 NaNfive 2002 Nevada 2.9 NaNIn [42]: frame2.columnsOut[42]: Index([year, state, pop, debt], dtype=object)

通过类似字典标记的方式或属性的方式，可以将DataFrame的列获取为一个Series：

In [43]: frame2['state'] In [44]: frame2.yearOut[43]: Out[44]:one Ohio one 2000two Ohio two 2001three Ohio three 2002four Nevada four 2001five Nevada five 2002Name: state Name: year

注意，返回的Series拥有原DataFrame相同的索引，且其name属性也已经被相应地设置好了。行也可以通过位置或名称的方式进行获取，比如用索引字段ix（稍后将对此进行详细讲解）：

In [45]: frame2.ix['three']Out[45]:year 2002state Ohiopop 3.6debt NaNName: three

列可以通过赋值的方式进行修改。例如，我们可以给那个空的"debt"列赋上一个标量值或一组值：

In [46]: frame2['debt'] = 16.5In [47]: frame2Out[47]: year state pop debtone 2000 Ohio 1.5 16.5two 2001 Ohio 1.7 16.5three 2002 Ohio 3.6 16.5four 2001 Nevada 2.4 16.5five 2002 Nevada 2.9 16.5In [48]: frame2['debt'] = np.arange(5.)In [49]: frame2Out[49]: year state pop debtone 2000 Ohio 1.5 0two 2001 Ohio 1.7 1three 2002 Ohio 3.6 2four 2001 Nevada 2.4 3five 2002 Nevada 2.9 4

将列表或数组赋值给某个列时，其长度必须跟DataFrame的长度相匹配。如果赋值的是一个Series，就会精确匹配DataFrame的索引，所有的空位都将被填上缺失值：

In [50]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])In [51]: frame2['debt'] = valIn [52]: frame2Out[52]: year state pop debtone 2000 Ohio 1.5 NaNtwo 2001 Ohio 1.7 -1.2three 2002 Ohio 3.6 NaNfour 2001 Nevada 2.4 -1.5five 2002 Nevada 2.9 -1.7

为不存在的列赋值会创建出一个新列。关键字del用于删除列：

In [53]: frame2['eastern'] = frame2.state == 'Ohio'In [54]: frame2Out[54]: year state pop debt easternone 2000 Ohio 1.5 NaN Truetwo 2001 Ohio 1.7 -1.2 Truethree 2002 Ohio 3.6 NaN Truefour 2001 Nevada 2.4 -1.5 Falsefive 2002 Nevada 2.9 -1.7 FalseIn [55]: del frame2['eastern']In [56]: frame2.columnsOut[56]: Index([year, state, pop, debt], dtype=object)

警告： 通过索引方式返回的列只是相应数据的视图而已，并不是副本。因此，对返回的Series所做的任何就地修改全都会反映到源DataFrame上。通过Series的copy方法即可显式地复制列。

另一种常见的数据形式是嵌套字典（也就是字典的字典）：

In [57]: pop = {'Nevada': {2001: 2.4, 2002: 2.9}, ...: 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}

如果将它传给DataFrame，它就会被解释为：外层字典的键作为列，内层键则作为行索引：

In [58]: frame3 = DataFrame(pop)In [59]: frame3Out[59]: Nevada Ohio2000 NaN 1.52001 2.4 1.72002 2.9 3.6

当然，你也可以对该结果进行转置：

In [60]: frame3.TOut[60]: 2000 2001 2002Nevada NaN 2.4 2.9Ohio 1.5 1.7 3.6

内层字典的键会被合并、排序以形成最终的索引。如果显式指定了索引，则不会这样：

In [61]: DataFrame(pop, index=[2001, 2002, 2003])Out[61]: Nevada Ohio2001 2.4 1.72002 2.9 3.62003 NaN NaN

由Series组成的字典差不多也是一样的用法：

In [62]: pdata = {'Ohio': frame3['Ohio'][:-1], ...: 'Nevada': frame3['Nevada'][:2]}In [63]: DataFrame(pdata)Out[63]: Nevada Ohio2000 NaN 1.52001 2.4 1.7

表5-1列出了DataFrame构造函数所能接受的各种数据。

表5-1：可以输入给DataFrame构造器的数据

类型	说明
二维ndarray	数据矩阵，还可以传入行标和列标
由数组、列表或元组组成的字典	每个序列会变成DataFrame的一列。所有序列的长度必须相同
NumPy的结构化/记录数组	类似于“由数组组成的字典”
由Series组成的字典	每个Series会成为一列。如果没有显式指定索引，则各Series的索引会被合并成结果的行索引
由字典组成的字典	各内层字典会成为一列。键会被合并成结果的行索引，跟“由Series组成的字典”的情况一样
字典或Series的列表	各项将会成为DataFrame的一行。字典键或Series索引的并集将会成为DataFrame的列标
由列表或元组组成的列表	类似于“二维ndarray”
另一个DataFrame	该DataFrame的索引将会被沿用，除非显式指定了其他索引
NumPy的MaskedArray	类似于“二维ndarray”的情况，只是掩码值在结果DataFrame会变成NA/缺失值

如果设置了DataFrame的index和columns的name属性，则这些信息也会被显示出来：

In [64]: frame3.index.name = 'year'; frame3.columns.name = 'state'In [65]: frame3Out[65]:state Nevada Ohioyear2000 NaN 1.52001 2.4 1.72002 2.9 3.6

跟Series一样，values属性也会以二维ndarray的形式返回DataFrame中的数据：

In [66]: frame3.valuesOut[66]:array([[nan, 1.5], [2.4, 1.7], [2.9, 3.6]])

如果DataFrame各列的数据类型不同，则值数组的数据类型就会选用能兼容所有列的数据类型：

In [67]: frame2.valuesOut[67]:array([[2000, Ohio, 1.5, nan], [2001, Ohio, 1.7, -1.2], [2002, Ohio, 3.6, nan], [2001, Nevada, 2.4, -1.5], [2002, Nevada, 2.9, -1.7]], dtype=object)

索引对象

pandas的索引对象负责管理轴标签和其他元数据（比如轴名称等）。构建Series或DataFrame时，所用到的任何数组或其他序列的标签都会被转换成一个Index：

In [68]: obj = Series(range(3), index=['a', 'b', 'c'])In [69]: index = obj.indexIn [70]: indexOut[70]: Index([a, b, c], dtype=object)In [71]: index[1:]Out[71]: Index([b, c], dtype=object)

Index对象是不可修改的（immutable），因此用户不能对其进行修改：

In [72]: index[1] = 'd'---Exception Traceback (most recent call last) in <module>()----> 1 index[1] = 'd'/Users/wesm/code/pandas/pandas/core/index.pyc in **setitem** (self, key, value) 302 def **setitem**(self, key, value): 303 """Disable the setting of values."""--> 304 raise Exception(str(self.**class**) + ' object is immutable') 305 306 def **getitem**(self, key):Exception: object is immutable

不可修改性非常重要，因为这样才能使Index对象在多个数据结构之间安全共享：

In [73]: index = pd.Index(np.arange(3))In [74]: obj2 = Series([1.5, -2.5, 0], index=index)In [75]: obj2.index is indexOut[75]: True

表5-2列出了pandas库中内置的Index类。由于开发人员的不懈努力，Index甚至可以被继承从而实现特别的轴索引功能。

注意： 虽然大部分用户都不需要知道太多关于Index对象的细节，但它们确实是pandas数据模型的重要组成部分。

表5-2：pandas中主要的Index对象

类	说明
Index	最泛化的Index对象，将轴标签表示为一个由Python对象组成的NumPy数组
Int64Index	针对整数的特殊Index
MultiIndex	“层次化”索引对象，表示单个轴上的多层索引。可以看做由元组组成的数组
DatetimeIndex	存储纳秒级时间戳（用NumPy的datetime64类型表示）
PeriodIndex	针对Period数据（时间间隔）的特殊Index

除了长得像数组，Index的功能也类似一个固定大小的集合：

In [76]: frame3Out[76]:state Nevada Ohioyear2000 NaN 1.52001 2.4 1.72002 2.9 3.6In [77]: 'Ohio' in frame3.columnsOut[77]: TrueIn [78]: 2003 in frame3.indexOut[78]: False

每个索引都有一些方法和属性，它们可用于设置逻辑并回答有关该索引所包含的数据的常见问题。表5-3列出了这些函数。

表5-3：Index的方法和属性

方法	说明
append	连接另一个Index对象，产生一个新的Index
diff	计算差集，并得到一个Index
intersection	计算交集
union	计算并集
isin	计算一个指示各值是否都包含在参数集合中的布尔型数组
delete	删除索引i处的元素，并得到新的Index
drop	删除传入的值，并得到新的Index
insert	将元素插入到索引i处，并得到新的Index
is_monotonic	当各元素均大于等于前一个元素时，返回True
is_unique	当Index没有重复值时，返回True
unique	计算Index中唯一值的数组

基本功能

本节中，我将介绍操作Series和DataFrame中的数据的基本手段。后续章节将更加深入地挖掘pandas在数据分析和处理方面的功能。本书不是pandas库的详尽文档，主要关注的是最重要的功能，那些不大常用的内容（也就是那些更深奥的内容）就交给你自己去摸索吧。

重新索引

pandas对象的一个重要方法是reindex，其作用是创建一个适应新索引的新对象。以之前的一个简单示例来说：

In [79]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])In [80]: objOut[80]:d 4.5b 7.2a -5.3c 3.6

调用该Series的reindex将会根据新索引进行重排。如果某个索引值当前不存在，就引入缺失值：

In [81]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])In [82]: obj2Out[82]:a -5.3b 7.2c 3.6d 4.5e NaNIn [83]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)Out[83]:a -5.3b 7.2c 3.6d 4.5e 0.0

对于时间序列这样的有序数据，重新索引时可能需要做一些插值处理。method选项即可达到此目的，例如，使用ffill可以实现前向值填充：

In [84]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])In [85]: obj3.reindex(range(6), method='ffill')Out[85]:0 blue1 blue2 purple3 purple4 yellow5 yellow

表5-4列出了可用的method选项。其实我们有时需要比前向和后向填充更为精准的插值方式。

表5-4：reindex的（插值）method选项

参数	说明
ffill或pad	前向填充（或搬运）值
bfill或backfill	后向填充（或搬运）值

对于DataFrame，reindex可以修改（行）索引、列，或两个都修改。如果仅传入一个序列，则会重新索引行：

In [86]: frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'], ...: columns=['Ohio', 'Texas', 'California'])In [87]: frameOut[87]: Ohio Texas Californiaa 0 1 2c 3 4 5d 6 7 8In [88]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])In [89]: frame2Out[89]: Ohio Texas Californiaa 0 1 2b NaN NaN NaNc 3 4 5d 6 7 8

使用columns关键字即可重新索引列：

In [90]: states = ['Texas', 'Utah', 'California']In [91]: frame.reindex(columns=states)Out[91]: Texas Utah Californiaa 1 NaN 2c 4 NaN 5d 7 NaN 8

也可以同时对行和列进行重新索引，而插值则只能按行应用（即轴0）：

In [92]: frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill', ...: columns=states)Out[92]: Texas Utah Californiaa 1 NaN 2b 1 NaN 2c 4 NaN 5d 7 NaN 8

利用ix的标签索引功能，重新索引任务可以变得更简洁：

In [93]: frame.ix[['a', 'b', 'c', 'd'], states]Out[93]: Texas Utah Californiaa 1 NaN 2b NaN NaN NaNc 4 NaN 5d 7 NaN 8

表5-5列出了reindex函数的各参数及说明。

表5-5：reindex函数的参数

参数	说明
index	用作索引的新序列。既可以是Index实例，也可以是其他序列型的Python数据结构。Index会被完全使用，就像没有任何复制一样
method	插值（填充）方式，具体参数请参见表5-4
fill_value	在重新索引的过程中，需要引入缺失值时使用的替代值
limit	前向或后向填充时的最大填充量
level	在MultiIndex的指定级别上匹配简单索引，否则选取其子集
copy	默认为True，无论如何都复制；如果为False，则新旧相等就不复制

丢弃指定轴上的项

丢弃某条轴上的一个或多个项很简单，只要有一个索引数组或列表即可。由于需要执行一些数据整理和集合逻辑，所以drop方法返回的是一个在指定轴上删除了指定值的新对象：

In [94]: obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])In [95]: new_obj = obj.drop('c')In [96]: new_objOut[96]:a 0b 1d 3e 4In [97]: obj.drop(['d', 'c'])Out[97]:a 0b 1e 4

对于DataFrame，可以删除任意轴上的索引值：

In [98]: data = DataFrame(np.arange(16).reshape((4, 4)), ...: index=['Ohio', 'Colorado', 'Utah', 'New York'], ...: columns=['one', 'two', 'three', 'four'])In [99]: data.drop(['Colorado', 'Ohio'])Out[99]: one two three fourUtah 8 9 10 11New York 12 13 14 15In [100]: data.drop('two', axis=1) [101]: data.drop(['two', 'four'], axis=1)Out[100]: Out[101]: one three four one threeOhio 0 2 3 Ohio 0 2Colorado 4 6 7 Colorado 4 6Utah 8 10 11 Utah 8 10New York 12 14 15 New York 12 14

索引、选取和过滤

Series索引（obj[...]）的工作方式类似于NumPy数组的索引，只不过Series的索引值不只是整数。下面是几个例子：

In [102]: obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])In [103]: obj['b'] In [104]: obj[1]Out[103]: 1.0 Out[104]: 1.0In [105]: obj[2:4] In [106]: obj[['b', 'a', 'd']]Out[105]: Out[106]:c 2 b 1d 3 a 0 d 3In [107]: obj[[1, 3]] In [108]: obj[obj < 2]Out[107]: Out[108]:b 1 a 0d 3 b 1

利用标签的切片运算与普通的Python切片运算不同，其末端是包含的（inclusive）译注1：

In [109]: obj['b':'c']Out[109]:b 1c 2

设置的方式也很简单：

In [110]: obj['b':'c'] = 5In [111]: objOut[111]:a 0b 5c 5d 3

如你所见，对DataFrame进行索引其实就是获取一个或多个列：

In [112]: data = DataFrame(np.arange(16).reshape((4, 4)), ...: index=['Ohio', 'Colorado', 'Utah', 'New York'], ...: columns=['one', 'two', 'three', 'four'])In [113]: dataOut[113]: one two three fourOhio 0 1 2 3Colorado 4 5 6 7Utah 8 9 10 11New York 12 13 14 15In [114]: data['two'] In [115]: data[['three', 'one']]Out[114]: Out[115]:Ohio 1 three oneColorado 5 Ohio 2 0Utah 9 Colorado 6 4New York 13 Utah 10 8Name: two New York 14 12

这种索引方式有几个特殊的情况。首先通过切片或布尔型数组选取行：

In [116]: data[:2] In [117]: data[data['three'] > 5]Out[116]: Out[117]: one two three four one two three fourOhio 0 1 2 3 Colorado 4 5 6 7Colorado 4 5 6 7 Utah 8 9 10 11 New York 12 13 14 15

有些读者可能会认为这不太合乎逻辑，但这种语法的的确确来源于实践。另一种用法是通过布尔型DataFrame（比如下面这个由标量比较运算得出的）进行索引：

In [118]: data < 5Out[118]: one two three fourOhio True True True TrueColorado True False False FalseUtah False False False FalseNew York False False False FalseIn [119]: data[data < 5] = 0In [120]: dataOut[120]: one two three fourOhio 0 0 0 0Colorado 0 5 6 7Utah 8 9 10 11New York 12 13 14 15

这段代码的目的是使DataFrame在语法上更像ndarray。

为了在DataFrame的行上进行标签索引，我引入了专门的索引字段ix。它使你可以通过NumPy式的标记法以及轴标签从DataFrame中选取行和列的子集。之前曾提到过，这也是一种重新索引的简单手段：

In [121]: data.ix['Colorado', ['two', 'three']]Out[121]:two 5three 6Name: ColoradoIn [122]: data.ix[['Colorado', 'Utah'], [3, 0, 1]]Out[122]: four one twoColorado 7 0 5Utah 11 8 9In [123]: data.ix[2] In [124]: data.ix[:'Utah', 'two']Out[123]: Out[124]:one 8 Ohio 0two 9 Colorado 5three 10 Utah 9four 11 Name: twoName: UtahIn [125]: data.ix[data.three > 5, :3]Out[125]: one two threeColorado 0 5 6Utah 8 9 10New York 12 13 14

也就是说，对pandas对象中的数据的选取和重排方式有很多。表5-6简单总结了针对DataFrame数据的大部分选取和重排方式。在使用层次化索引时还能用到一些别的办法（稍后就会讲到）。

注意：在设计pandas时，我觉得必须输入frame[:,col]才能选取列实在有些啰嗦（而且还很容易出错），因为列的选取是一种最常见的操作。于是，我就把所有的标签索引功能都放到ix中了。

表5-6：DataFrame的索引选项

类型	说明
obj[val]	选取DataFrame的单个列或一组列。在一些特殊情况下会比较便利：布尔型数组（过滤行）、切片（行切片）、布尔型DataFrame（根据条件设置值）
obj.ix[val]	选取DataFrame的单个行或一组行

表5-6：DataFrame的索引选项（续）

类型	说明
obj.ix[:, val]	选取单个列或列子集
obj.ix[val1, val2]	同时选取行和列
reindex方法	将一个或多个轴匹配到新索引
xs方法	根据标签选取单行或单列，并返回一个Series
icol、irow方法	根据整数位置选取单列或单行，并返回一个Series
get_value、set_value方法	根据行标签和列标签选取单个值。译注2

译注2：get_value方法是选取，set-value方法是设置。

算术运算和数据对齐

pandas最重要的一个功能是，它可以对不同索引的对象进行算术运算。在将对象相加时，如果存在不同的索引对，则结果的索引就是该索引对的并集。我们来看一个简单的例子：

In [126]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])In [127]: s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])In [128]: s1 In [129]: s2Out[128]: Out[129]:a 7.3 a -2.1c -2.5 c 3.6d 3.4 e -1.5e 1.5 f 4.0 g 3.1

将它们相加就会产生：

In [130]: s1 + s2Out[130]:a 5.2c 1.1d NaNe 0.0f NaNg NaN

自动的数据对齐操作在不重叠的索引处引入了NA值译注3。缺失值会在算术运算过程中传播。

对于DataFrame，对齐操作会同时发生在行和列上：

In [131]: df1 = DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'), ...: index=['Ohio', 'Texas', 'Colorado'])In [132]: df2 = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'), ...: index=['Utah', 'Ohio', 'Texas', 'Oregon'])In [133]: df1 In [134]: df2Out[133]: Out[134]: b c d b d eOhio 0 1 2 Utah 0 1 2Texas 3 4 5 Ohio 3 4 5Colorado 6 7 8 Texas 6 7 8 Oregon 9 10 11

把它们相加后将会返回一个新的DataFrame，其索引和列为原来那两个DataFrame的并集：

In [135]: df1 + df2Out[135]: b c d eColorado NaN NaN NaN NaNOhio 3 NaN 6 NaNOregon NaN NaN NaN NaNTexas 9 NaN 12 NaNUtah NaN NaN NaN NaN

在算术方法中填充值

在对不同索引的对象进行算术运算时，你可能希望当一个对象中某个轴标签在另一个对象中找不到时填充一个特殊值（比如0）：

In [136]: df1 = DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))In [137]: df2 = DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))In [138]: df1 In [139]: df2Out[138]: Out[139]: a b c d a b c d e0 0 1 2 3 0 0 1 2 3 41 4 5 6 7 1 5 6 7 8 92 8 9 10 11 2 10 11 12 13 14 3 15 16 17 18 19

将它们相加时，没有重叠的位置就会产生NA值：

In [140]: df1 + df2Out[140]: a b c d e0 0 2 4 6 NaN1 9 11 13 15 NaN2 18 20 22 24 NaN3 NaN NaN NaN NaN NaN

使用df1的add方法，传入df2以及一个fill_value参数：

In [141]: df1.add(df2, fill_value=0)Out[141]: a b c d e0 0 2 4 6 41 9 11 13 15 92 18 20 22 24 143 15 16 17 18 19

与此类似，在对Series或DataFrame重新索引时，也可以指定一个填充值：

In [142]: df1.reindex(columns=df2.columns, fill_value=0)Out[142]: a b c d e0 0 1 2 3 01 4 5 6 7 02 8 9 10 11 0

表5-7：灵活的算术方法

方法	说明
add	用于加法（+）的方法
sub	用于减法（－）的方法
div	用于除法（/）的方法
mul	用于乘法（*）的方法

DataFrame和Series之间的运算

跟NumPy数组一样，DataFrame和Series之间算术运算也是有明确规定的。先来看一个具有启发性的例子，计算一个二维数组与其某行之间的差：

In [143]: arr = np.arange(12.).reshape((3, 4))In [144]: arrOut[144]:array([[0., 1., 2., 3.], [4., 5., 6., 7.], [8., 9., 10., 11.]])In [145]: arr[0]Out[145]: array([0., 1., 2., 3.])In [146]: arr - arr[0]Out[146]:array([[0., 0., 0., 0.], [4., 4., 4., 4.], [8., 8., 8., 8.]])

这就叫做广播（broadcasting），第12章将对此进行详细讲解。DataFrame和Series之间的运算差不多也是如此：

In [147]: frame = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'), ...: index=['Utah', 'Ohio', 'Texas', 'Oregon'])In [148]: series = frame.ix[0]In [149]: frame In [150]: seriesOut[149]: Out[150]: b d e b 0Utah 0 1 2 d 1Ohio 3 4 5 e 2Texas 6 7 8 Name: UtahOregon 9 10 11

默认情况下，DataFrame和Series之间的算术运算会将Series的索引匹配到DataFrame的列，然后沿着行一直向下广播：

In [151]: frame - seriesOut[151]: b d eUtah 0 0 0Ohio 3 3 3Texas 6 6 6Oregon 9 9 9

如果某个索引值在DataFrame的列或Series的索引中找不到，则参与运算的两个对象就会被重新索引以形成并集：

In [152]: series2 = Series(range(3), index=['b', 'e', 'f'])In [153]: frame + series2Out[153]: b d e fUtah 0 NaN 3 NaNOhio 3 NaN 6 NaNTexas 6 NaN 9 NaNOregon 9 NaN 12 NaN

如果你希望匹配行且在列上广播，则必须使用算术运算方法。例如：

In [154]: series3 = frame['d']In [155]: frame In [156]: series3Out[155]: Out[156]: b d e Utah 1Utah 0 1 2 Ohio 4Ohio 3 4 5 Texas 7Texas 6 7 8 Oregon 10Oregon 9 10 11 Name: dIn [157]: frame.sub(series3, axis=0)Out[157]: b d eUtah -1 0 1Ohio -1 0 1Texas -1 0 1Oregon -1 0 1

传入的轴号就是希望匹配的轴。在本例中，我们的目的是匹配DataFrame的行索引并进行广播。译注4

函数应用和映射

NumPy的ufuncs（元素级数组方法）也可用于操作pandas对象：

In [158]: frame = DataFrame(np.random.randn(4, 3), columns=list('bde'), ...: index=['Utah', 'Ohio', 'Texas', 'Oregon'])In [159]: frame In [160]: np.abs(frame)Out[159]: Out[160]: b d e b d eUtah -0.204708 0.478943 -0.519439 Utah 0.204708 0.478943 0.519439Ohio -0.555730 1.965781 1.393406 Ohio 0.555730 1.965781 1.393406Texas 0.092908 0.281746 0.769023 Texas 0.092908 0.281746 0.769023Oregon 1.246435 1.007189 -1.296221 Oregon 1.246435 1.007189 1.296221

另一个常见的操作是，将函数应用到由各列或行所形成的一维数组上。DataFrame的apply方法即可实现此功能：

In [161]: f = lambda x: x.max() - x.min()In [162]: frame.apply(f) In [163]: frame.apply(f, axis=1)Out[162]: Out[163]:b 1.802165 Utah 0.998382d 1.684034 Ohio 2.521511e 2.689627 Texas 0.676115 Oregon 2.542656

许多最为常见的数组统计功能都被实现成DataFrame的方法（如sum和mean），因此无需使用apply方法。

除标量值外，传递给apply的函数还可以返回由多个值组成的Series：

In [164]: def f(x): ...: return Series([x.min(), x.max()], index=['min', 'max'])In [165]: frame.apply(f) b d emin -0.555730 0.281746 -1.296221max 1.246435 1.965781 1.393406

此外，元素级的Python函数也是可以用的。假如你想得到frame中各个浮点值的格式化字符串，使用applymap即可：

In [166]: format = lambda x: '%.2f' % xIn [167]: frame.applymap(format)Out[167]: b d eUtah -0.20 0.48 -0.52Ohio -0.56 1.97 1.39Texas 0.09 0.28 0.77Oregon 1.25 1.01 -1.30

之所以叫做applymap，是因为Series有一个用于应用元素级函数的map方法：

In [168]: frame['e'].map(format)Out[168]:Utah -0.52Ohio 1.39Texas 0.77Oregon -1.30Name: e

排序和排名

根据条件对数据集排序（sorting）也是一种重要的内置运算。要对行或列索引进行排序（按字典顺序），可使用sort_index方法，它将返回一个已排序的新对象：

In [169]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])In [170]: obj.sort_index()Out[170]:a 1b 2c 3d 0

而对于DataFrame，则可以根据任意一个轴上的索引进行排序：

In [171]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'], ...: columns=['d', 'a', 'b', 'c'])In [172]: frame.sort_index() In [173]: frame.sort_index(axis=1)Out[172]: Out[173]: d a b c a b c done 4 5 6 7 three 1 2 3 0three 0 1 2 3 one 5 6 7 4

数据默认是按升序排序的，但也可以降序排序：

In [174]: frame.sort_index(axis=1, ascending=False)Out[174]: d c b athree 0 3 2 1one 4 7 6 5

若要按值对Series进行排序，可使用其order方法：

In [175]: obj = Series([4, 7, -3, 2])In [176]: obj.order()Out[176]:2 -33 20 41 7

在排序时，任何缺失值默认都会被放到Series的末尾：

In [177]: obj = Series([4, np.nan, 7, np.nan, -3, 2])In [178]: obj.order()Out[178]:4 -35 20 42 71 NaN3 NaN

在DataFrame上，你可能希望根据一个或多个列中的值进行排序。将一个或多个列的名字传递给by选项即可达到该目的：

In [179]: frame = DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})In [180]: frame In [181]: frame.sort_index(by='b')Out[180]: Out[181]: a b a b0 0 4 2 0 -31 1 7 3 1 22 0 -3 0 0 43 1 2 1 1 7

要根据多个列进行排序，传入名称的列表即可：

In [182]: frame.sort_index(by=['a', 'b'])Out[182]: a b2 0 -30 0 43 1 21 1 7

排名（ranking）跟排序关系密切，且它会增设一个排名值（从1开始，一直到数组中有效数据的数量）。它跟numpy.argsort产生的间接排序索引差不多，只不过它可以根据某种规则破坏平级关系。接下来介绍Series和DataFrame的rank方法。默认情况下，rank是通过“为各组分

配一个平均排名”的方式破坏平级关系的：

In [183]: obj = Series([7, -5, 7, 4, 2, 0, 4])In [184]: obj.rank()Out[184]:0 6.51 1.02 6.53 4.54 3.05 2.06 4.5

也可以根据值在原数据中出现的顺序给出排名译注5：

In [185]: obj.rank(method='first')Out[185]:0 61 12 73 44 35 26 5

当然，你也可以按降序进行排名：

In [186]: obj.rank(ascending=False, method='max')Out[186]:0 21 72 23 44 55 66 4

表5-8列出了所有用于破坏平级关系的method选项。DataFrame可以在行或列上计算排名：

In [187]: frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1], ...: 'c': [-2, 5, 8, -2.5]})In [188]: frame In [189]: frame.rank(axis=1)Out[188]: Out[189]: a b c a b c0 0 4.3 -2.0 0 2 3 11 1 7.0 5.0 1 1 3 22 0 -3.0 8.0 2 2 1 33 1 2.0 -2.5 3 2 3 1

表5-8：排名时用于破坏平级关系的method选项

method	说明
'average'	默认：在相等分组中，为各个值分配平均排名
'min'	使用整个分组的最小排名
'max'	使用整个分组的最大排名
'first'	按值在原始数据中的出现顺序分配排名

带有重复值的轴索引

直到目前为止，我所介绍的所有范例都有着唯一的轴标签（索引值）。虽然许多pandas函数（如reindex）都要求标签唯一，但这并不是强制性的。我们来看看下面这个简单的带有重复索引值的Series：

In [190]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])In [191]: objOut[191]:a 0a 1b 2b 3c 4

索引的is_unique属性可以告诉你它的值是否是唯一的：

In [192]: obj.index.is_uniqueOut[192]: False

对于带有重复值的索引，数据选取的行为将会有些不同。如果某个索引对应多个值，则返回一个Series；而对应单个值的，则返回一个标量值。

In [193]: obj['a'] In [194]: obj['c']Out[193]: Out[194]: 4a 0a 1

对DataFrame的行进行索引时也是如此：

In [195]: df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])In [196]: dfOut[196]: 0 1 2a 0.274992 0.228913 1.352917a 0.886429 -2.001637 -0.371843b 1.669025 -0.438570 -0.539741b 0.476985 3.248944 -1.021228In [197]: df.ix['b']Out[197]: 0 1 2b 1.669025

-0.438570 -0.539741b 0.476985 3.248944 -1.021228

译注1

：即封闭区间。

译注3

：由于本书中多次出现“非重叠”（overlapping）这个词，所以需要简单说明一下。例如，“飞机场”跟“拖拉机”都有个“机”，于是可以认为这两个字符串是“重叠”的；“高富帅”和“矮穷挫”的情况自然就是“非重叠”了。注意，虽然这里没有任何顺序和连续的概念，但有些地方是需要考虑顺序和连续的。

译注4

：这里需要补充说明一下，作者反复强调“广播”会在第12章介绍，所以如果真看不懂这里就等到12章学完再看不迟。译者已经尽量把原文扩展的描述扩展开，但是文字描述始终没有图形更具体。例如，你可以打开一个Excel，随意找一排单元格并输入一些文字（注意是一排），然后选中这些单元格，将鼠标移至选区右下角，当指针变为加号时，按住向下拉几行，这就是“沿行向下广播”。

译注5

：类似于稳定排序。

汇总和计算描述统计

pandas对象拥有一组常用的数学和统计方法。它们大部分都属于约简和汇总统计，用于从Series中提取单个值（如sum或mean）或从DataFrame的行或列中提取一个Series。跟对应的NumPy数组方法相比，它们都是基于没有缺失数据的假设而构建的。接下来看一个简单的DataFrame：

In [198]: df = DataFrame([[1.4, np.nan], [7.1, -4.5], ...: [np.nan, np.nan], [0.75, -1.3]], ...: index=['a', 'b', 'c', 'd'], ...: columns=['one', 'two'])In [199]: dfOut[199]: one twoa 1.40 NaNb 7.10 -4.5c NaN NaNd 0.75 -1.3

调用DataFrame的sum方法将会返回一个含有列小计的Series：

In [200]: df.sum()Out[200]:one 9.25two -5.80

传入axis=1将会按行进行求和运算：

In [201]: df.sum(axis=1)Out[201]:a 1.40b 2.60c NaNd -0.55

NA值会自动被排除，除非整个切片（这里指的是行或列）都是NA。通过skipna选项可以禁用该功能：

In [202]: df.mean(axis=1, skipna=False)Out[202]:a NaNb 1.300c NaNd -0.275

表5-9列出了这些约简方法的常用选项。

表5-9：约简方法的选项

选项	说明
axis	约简的轴。DataFrame的行用0，列用1
skipna	排除缺失值，默认值为True
level	如果轴是层次化索引的（即MultiIndex），则根据level分组约简

有些方法（如idxmin和idxmax）返回的是间接统计（比如达到最小值或最大值的索引）：

In [203]: df.idxmax()Out[203]:one btwo d

另一些方法则是累计型的：

In [204]: df.cumsum()Out[204]: one twoa 1.40 NaNb 8.50 -4.5c NaN NaNd 9.25 -5.8

还有一种方法，它既不是约简型也不是累计型。describe就是一个例子，它用于一次性产生多个汇总统计：

In [205]: df.describe()Out[205]: one twocount 3.000000 2.000000mean 3.083333 -2.900000std 3.493685 2.262742min 0.750000 -4.50000025% 1.075000 -3.70000050% 1.400000 -2.90000075% 4.250000 -2.100000max 7.100000 -1.300000

对于非数值型数据，describe会产生另外一种汇总统计：

In [206]: obj = Series(['a', 'a', 'b', 'c'] * 4)In [207]: obj.describe()Out[207]:count 16unique 3top afreq 8

表5-10列出了所有与描述统计相关的方法。

表5-10：描述和汇总统计

方法	说明
count	非NA值的数量
describe	针对Series或各DataFrame列计算汇总统计
min、max	计算最小值和最大值
argmin、argmax	计算能够获取到最小值和最大值的索引位置（整数）
idxmin、idxmax	计算能够获取到最小值和最大值的索引值
quantile	计算样本的分位数（0到1）
sum	值的总和
mean	值的平均数
median	值的算术中位数（50%分位数）
mad	根据平均值计算平均绝对离差
var	样本值的方差
std	样本值的标准差

表5-10：描述和汇总统计（续）

方法	说明
skew	样本值的偏度（三阶矩）
kurt	样本值的峰度（四阶矩）
cumsum	样本值的累计和
cummin、cummax	样本值的累计最大值和累计最小值
cumprod	样本值的累计积
diff	计算一阶差分（对时间序列很有用）
pct_change	计算百分数变化

相关系数与协方差

有些汇总统计（如相关系数和协方差）是通过参数对计算出来的。我们来看几个DataFrame，它们的数据来自Yahoo!Finance的股票价格和成交量：

import pandas.io.data as weball_data = {}for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']: all_data[ticker] = web.get_data_yahoo(ticker, '1/1/2000', '1/1/2010')price = DataFrame({tic: data['Adj Close'] for tic, data in all_data.iteritems()})volume = DataFrame({tic: data['Volume'] for tic, data in all_data.iteritems()})

接下来计算价格的百分数变化：

In [209]: returns = price.pct_change()In [210]: returns.tail()Out[210]: AAPL GOOG IBM MSFTDate2009-12-24 0.034339 0.011117 0.004420 0.0027472009-12-28 0.012294 0.007098 0.013282 0.0054792009-12-29 -0.011861 -0.005571 -0.003474 0.0068122009-12-30 0.012147 0.005376 0.005468 -0.0135322009-12-31 -0.004300 -0.004416 -0.012609 -0.015432

Series的corr方法用于计算两个Series中重叠的、非NA的、按索引对齐的值的相关系数。与此类似，cov用于计算协方差：

In [211]: returns.MSFT.corr(returns.IBM)Out[211]: 0.49609291822168838In [212]: returns.MSFT.cov(returns.IBM)Out[212]: 0.00021600332437329015

DataFrame的corr和cov方法将以DataFrame的形式返回完整的相关系数或协方差矩阵：

In [213]: returns.corr()Out[213]: AAPL GOOG IBM MSFTAAPL 1.000000 0.470660 0.410648 0.424550GOOG 0.470660 1.000000 0.390692 0.443334IBM 0.410648 0.390692 1.000000 0.496093MSFT 0.424550 0.443334 0.496093 1.000000In [214]: returns.cov()Out[214]: AAPL GOOG IBM MSFTAAPL 0.001028 0.000303 0.000252 0.000309GOOG 0.000303 0.000580 0.000142 0.000205IBM 0.000252 0.000142 0.000367 0.000216MSFT 0.000309 0.000205 0.000216 0.000516

利用DataFrame的corrwith方法，你可以计算其列或行跟另一个Series或DataFrame之间的相关系数。传入一个Series将会返回一个相关系数值Series（针对各列进行计算）：

In [215]: returns.corrwith(returns.IBM)Out[215]:AAPL 0.410648GOOG 0.390692IBM 1.000000MSFT 0.496093

传入一个DataFrame则会计算按列名配对的相关系数。这里，我计算百分比变化与成交量的相关系数：

In [216]: returns.corrwith(volume)Out[216]:AAPL -0.057461GOOG 0.062644IBM -0.007900MSFT -0.014175

传入axis=1即可按行进行计算。无论如何，在计算相关系数之前，所有的数据项都会按标签对齐。

唯一值、值计数以及成员资格

还有一类方法可以从一维Series的值中抽取信息。以下面这个Series为例：

In [217]: obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])

第一个函数是unique，它可以得到Series中的唯一值数组：

In [218]: uniques = obj.unique()In [219]: uniquesOut[219]: array([c, a, d, b], dtype=object)

返回的唯一值是未排序的，如果需要的话，可以对结果再次进行排序（uniques.sort()）。value_counts用于计算一个Series中各值出现的频率：

In [220]: obj.value_counts()Out[220]:c 3a 3b 2d 1

为了便于查看，结果Series是按值频率降序排列的。value_counts还是一个顶级pandas方法，可用于任何数组或序列：

In [221]: pd.value_counts(obj.values, sort=False)Out[221]:a 3b 2c 3d 1

最后是isin，它用于判断矢量化集合的成员资格，可用于选取Series中或DataFrame列中数据的子集：

In [222]: mask = obj.isin(['b', 'c'])In [223]: mask In [224]: obj[mask]Out[223]: Out[224]:0 True 0 c1 False 5 b2 False 6 b3 False 7 c4 False 8 c5 True6 True7 True8 True

表5-11给出了这几个方法的一些参考信息。

表5-11：唯一值、值计数、成员资格方法

方法	说明
isin	计算一个表示“Series各值是否包含于传入的值序列中”的布尔型数组
unique	计算Series中的唯一值数组，按发现的顺序返回
value_counts	返回一个Series，其索引为唯一值，其值为频率，按计数值降序排列

有时，你可能希望得到DataFrame中多个相关列的一张柱状图。例如：

In [225]: data = DataFrame({'Qu1': [1, 3, 4, 3, 4], ...: 'Qu2': [2, 3, 1, 2, 3], ...: 'Qu3': [1, 5, 2, 4, 4]})In [226]: dataOut[226]: Qu1 Qu2 Qu30 1 2 11 3 3 52 4 1 23 3 2 44 4 3 4

将pandas.value_counts传给该DataFrame的apply函数，就会出现：

In [227]: result = data.apply(pd.value_counts).fillna(0)In [228]: resultOut[228]: Qu1 Qu2 Qu31 1 1 12 0 2 13 2 2 04 2 0 25 0 0 1

处理缺失数据

缺失数据（missing data）在大部分数据分析应用中都很常见。pandas的设计目标之一就是让缺失数据的处理任务尽量轻松。例如，pandas对象上的所有描述统计都排除了缺失数据，正如我们在本章稍早的地方所看到的那样。

pandas使用浮点值NaN（Not a Number）表示浮点和非浮点数组中的缺失数据。它只是一个便于被检测出来的标记而已：

In [229]: string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])In [230]: string_data In [231]: string_data.isnull()Out[230]: Out[231]:0 aardvark 0 False1 artichoke 1 False2 NaN 2 True3 avocado 3 False

Python内置的None值也会被当做NA处理：

In [232]: string_data[0] = NoneIn [233]: string_data.isnull()Out[233]:0 True1 False2 True3 False

我不敢说pandas的NA表现形式是最优的，但它确实很简单也很可靠。由于NumPy的数据类型体系中缺乏真正的NA数据类型或位模式，所以它是我能想到的最佳解决方案（一套简单的API以及足够全面的性能特征）。随着NumPy的不断发展，这个问题今后可能会发生变化。

表5-12：NA处理方法

方法	说明
dropna	根据各标签的值中是否存在缺失数据对轴标签进行过滤，可通过阈值调节对缺失值的容忍度
fillna	用指定值或插值方法（如ffill或bfill）填充缺失数据
isnull	返回一个含有布尔值的对象，这些布尔值表示哪些值是缺失值/NA，该对象的类型与源类型一样
notnull	isnull的否定式

滤除缺失数据

过滤掉缺失数据的办法有很多种。纯手工操作永远都是一个办法，但dropna可能会更实用一些。对于一个Series，dropna返回一个仅含非空数据和索引值的Series：

In [234]: from numpy import nan as NAIn [235]: data = Series([1, NA, 3.5, NA, 7])In [236]: data.dropna()Out[236]:0 1.02 3.54 7.0

当然，也可以通过布尔型索引达到这个目的：

In [237]: data[data.notnull()]Out[237]:0 1.02 3.54 7.0

而对于DataFrame对象，事情就有点复杂了。你可能希望丢弃全NA或含有NA的行或列。dropna默认丢弃任何含有缺失值的行：

In [238]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA], ...: [NA, NA, NA], [NA, 6.5, 3.]])In [239]: cleaned = data.dropna()In [240]: data In [241]: cleanedOut[240]: Out[241]: 0 1 2 0 1 20 1 6.5 3 0 1 6.5 31 1 NaN NaN2 NaN NaN NaN3 NaN 6.5 3

传入how='all'将只丢弃全为NA的那些行：

In [242]: data.dropna(how='all')Out[242]: 0 1 20 1 6.5 31 1 NaN NaN3 NaN 6.5 3

要用这种方式丢弃列，只需传入axis=1即可：

In [243]: data[4] = NAIn [244]: data In [245]: data.dropna(axis=1, how='all')Out[244]: Out[245]: 0 1 2 4 0 1 20 1 6.5 3 NaN 0 1 6.5 31 1 NaN NaN NaN 1 1 NaN NaN2 NaN NaN NaN NaN 2 NaN NaN NaN3 NaN 6.5 3 NaN 3 NaN 6.5 3

另一个滤除DataFrame行的问题涉及时间序列数据。假设你只想留下一部分观测数据，可以用thresh参数实现此目的：

In [246]: df = DataFrame(np.random.randn(7, 3))In [247]: df.ix[:4, 1] = NA; df.ix[:2, 2] = NAIn [248]: df In [249]: df.dropna(thresh=3)Out[248]: Out[249]: 0 1 2 0 1 20 -0.577087 NaN NaN 5 0.332883 -2.359419 -0.1995431 0.523772 NaN NaN 6-1.541996 -0.970736 -1.3070302 -0.713544 NaN NaN3 -1.860761 NaN 0.5601454 -1.265934 NaN -1.0635125 0.332883 -2.359419 -0.1995436 -1.541996 -0.970736 -1.307030

填充缺失数据

你可能不想滤除缺失数据（有可能会丢弃跟它有关的其他数据），而是希望通过其他方式填补那些“空洞”。对于大多数情况而言，fillna方法是最主要的函数。通过一个常数调用fillna就会将缺失值替换为那个常数值：

In [250]: df.fillna(0)Out[250]: 0 1 20 -0.577087 0.000000 0.0000001 0.523772 0.000000 0.0000002 -0.713544 0.000000 0.0000003 -1.860761 0.000000 0.5601454 -1.265934 0.000000 -1.0635125 0.332883 -2.359419 -0.1995436 -1.541996 -0.970736 -1.307030

若是通过一个字典调用fillna，就可以实现对不同的列填充不同的值：

In [251]: df.fillna({1: 0.5, 3: -1})Out[251]: 0 1 20 -0.577087 0.500000 NaN1 0.523772 0.500000 NaN2 -0.713544 0.500000 NaN3 -1.860761 0.500000 0.5601454 -1.265934 0.500000 -1.0635125 0.332883 -2.359419 -0.1995436 -1.541996 -0.970736 -1.307030

fillna默认会返回新对象，但也可以对现有对象进行就地修改：

总是返回被填充对象的引用In [252]: _ = df.fillna(0, inplace=True)In [253]: dfOut[253]: 0 1 20 -0.577087 0.000000 0.0000001 0.523772 0.000000 0.0000002 -0.713544 0.000000 0.0000003 -1.860761 0.000000 0.5601454 -1.265934 0.000000 -1.0635125 0.332883 -2.359419 -0.1995436 -1.541996 -0.970736 -1.307030

对reindex有效的那些插值方法也可用于fillna：

In [254]: df = DataFrame(np.random.randn(6, 3))In [255]: df.ix[2:, 1] = NA; df.ix[4:, 2] = NAIn [256]: dfOut[256]: 0 1 20 0.286350 0.377984 -0.7538871 0.331286 1.349742 0.0698772 0.246674 NaN 1.0048123 1.327195 NaN -1.5491064 0.022185 NaN NaN5 0.862580 NaN NaNIn [257]: df.fillna(method='ffill') In [258]: df.fillna(method='ffill', limit=2)Out[257]: Out[258]: 0 1 2 0 1 20 0.286350 0.377984 -0.753887 0 0.286350 0.377984 -0.7538871 0.331286 1.349742 0.069877 1 0.331286 1.349742 0.0698772 0.246674 1.349742 1.004812 2 0.246674 1.349742 1.0048123 1.327195 1.349742 -1.549106 3 1.327195 1.349742 -1.5491064 0.022185 1.349742 -1.549106 4 0.022185 NaN -1.5491065 0.862580 1.349742 -1.549106 5 0.862580 NaN -1.549106

只要稍微动动脑子，你就可以利用fillna实现许多别的功能。比如说，你可以传入Series的平均值或中位数：

In [259]: data = Series([1., NA, 3.5, NA, 7])In [260]: data.fillna(data.mean())Out[260]:0 1.0000001 3.8333332 3.5000003 3.8333334 7.000000

表5-13列出了fillna的参数参考。

表5-13：fillna函数的参数

参数	说明
value	用于填充缺失值的标量值或字典对象
method	插值方式。如果函数调用时未指定其他参数的话，默认为“ffill”

表5-13：fillna函数的参数（续）

参数	说明
axis	待填充的轴，默认axis=0
inplace	修改调用者对象而不产生副本
limit	（对于前向和后向填充）可以连续填充的最大数量

层次化索引

层次化索引（hierarchical indexing）是pandas的一项重要功能，它使你能在一个轴上拥有多个（两个以上）索引级别。抽象点说，它使你能以低维度形式处理高维度数据。我们先来看一个简单的例子：创建一个Series，并用一个由列表或数组组成的列表作为索引。

In [261]: data = Series(np.random.randn(10), ...: index=[['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'd', 'd'], ...: [1, 2, 3, 1, 2, 3, 1, 2, 2, 3]])In [262]: dataOut[262]:a 1 0.670216 2 0.852965 3 -0.955869b 1 -0.023493 2 -2.304234 3 -0.652469c 1 -1.218302 2 -1.332610d 2 1.074623 3 0.723642

这就是带有MultiIndex索引的Series的格式化输出形式。索引之间的“间隔”表示“直接使用上面的标签”：

In [263]: data.indexOut[263]:MultiIndex[('a', 1) ('a', 2) ('a', 3) ('b', 1) ('b', 2) ('b', 3) ('c', 1) ('c', 2) ('d', 2) ('d', 3)]

对于一个层次化索引的对象，选取数据子集的操作很简单：

In [264]: data['b']Out[264]:1 -0.0234932 -2.3042343 -0.652469In [265]: data['b':'c'] In [266]: data.ix[['b', 'd']]Out[265]: Out[266]:b 1 -0.023493 b 1 -0.023493 2 -2.304234 2 -2.304234 3 -0.652469 3 -0.652469c 1 -1.218302 d 2 1.074623 2 -1.332610 3 0.723642

有时甚至还可以在“内层”中进行选取：

In [267]: data[:, 2]Out[267]:a 0.852965b -2.304234c -1.332610d 1.074623

层次化索引在数据重塑和基于分组的操作（如透视表生成）中扮演着重要的角色。比如说，这段数据可以通过其unstack方法被重新安排到一个DataFrame中：

In [268]: data.unstack()Out[268]: 1 2 3a 0.670216 0.852965 -0.955869b -0.023493 -2.304234 -0.652469c -1.218302 -1.332610 NaNd NaN 1.074623 0.723642

unstack的逆运算是stack：

In [269]: data.unstack().stack()Out[269]:a 1 0.670216 2 0.852965 3 -0.955869b 1 -0.023493 2 -2.304234 3 -0.652469c 1 -1.218302 2 -1.332610d 2 1.074623 3 0.723642

stack和unstack将在第7章中详细讲解。

对于一个DataFrame，每条轴都可以有分层索引：

In [270]: frame = DataFrame(np.arange(12).reshape((4, 3)), ...: index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]], ...: columns=[['Ohio', 'Ohio', 'Colorado'], ...: ['Green', 'Red', 'Green']])In [271]: frameOut[271]: Ohio Colorado Green Red Greena 1 0 1 2 2 3 4 5b 1 6 7 8 2 9 10 11

各层都可以有名字（可以是字符串，也可以是别的Python对象）。如果指定了名称，它们就会显示在控制台输出中（不要将索引名称跟轴标签混为一谈！）：

In [272]: frame.index.names = ['key1', 'key2']In [273]: frame.columns.names = ['state', 'color']In [274]: frameOut[274]:state Ohio Coloradocolor Green Red Greenkey1 key2a 1 0 1 2 2 3 4 5b 1 6 7 8 2 9 10 11

由于有了分部的列索引，因此可以轻松选取列分组：

In [275]: frame['Ohio']Out[275]:color Green Redkey1 key2a 1 0 1 2 3 4b 1 6 7 2 9 10

可以单独创建MultiIndex然后复用。上面那个DataFrame中的（分级的）列可以这样创建：

MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']], names=['state', 'color'])

重排分级顺序

有时，你需要重新调整某条轴上各级别的顺序，或根据指定级别上的值对数据进行排序。swaplevel接受两个级别编号或名称，并返回一个互换了级别的新对象（但数据不会发生变化）：

In [276]: frame.swaplevel('key1', 'key2')Out[276]:state Ohio Coloradocolor Green Red Greenkey2 key11 a 0 1 22 a 3 4 51 b 6 7 82 b 9 10 11

而sortlevel则根据单个级别中的值对数据进行排序（稳定的）。交换级别时，常常也会用到sortlevel，这样最终结果就是有序的了：

In [277]: frame.sortlevel(1) In [278]: frame.swaplevel(0, 1).sortlevel(0)Out[277]: Out[278]:state Ohio Colorado state Ohio Coloradocolor Green Red Green color Green Red Greenkey1 key2 key2 key1a 1 0 1 2 1 a 0 1 2b 1 6 7 8 b 6 7 8a 2 3 4 5 2 a 3 4 5b 2 9 10 11 b 9 10 11

注意：在层次化索引的对象上，如果索引是按字典方式从外到内排序（即调用sortlevel(0)或sort_index()的结果），数据选取操作的性能要好很多。

根据级别汇总统计

许多对DataFrame和Series的描述和汇总统计都有一个level选项，它用于指定在某条轴上求和的级别。再以上面那个DataFrame为例，我们可以根据行或列上的级别来进行求和，如下所示：

In [279]: frame.sum(level='key2')Out[279]:state Ohio Coloradocolor Green Red Greenkey21 6 8 102 12 14 16In [280]: frame.sum(level='color', axis=1)Out[280]:color Green Redkey1 key2a 1 2 1 2 8 4b 1 14 7 2 20 10

这其实是利用了pandas的groupby功能，本书稍后将对其进行详细讲解。

使用DataFrame的列

人们经常想要将DataFrame的一个或多个列当做行索引来用，或者可能希望将行索引变成DataFrame的列。以下面这个DataFrame为例：

In [281]: frame = DataFrame({'a': range(7), 'b': range(7, 0, -1), ...: 'c': ['one', 'one', 'one', 'two', 'two', 'two', 'two'], ...: 'd': [0, 1, 2, 0, 1, 2, 3]})In [282]: frameOut[282]: a b c d0 0 7 one 01 1 6 one 12 2 5 one 23 3 4 two 04 4 3 two 15 5 2 two 26 6 1 two 3

DataFrame的set_index函数会将其一个或多个列转换为行索引，并创建一个新的DataFrame：

In [283]: frame2 = frame.set_index(['c', 'd'])In [284]: frame2Out[284]: a bc done 0 0 7 1 1 6 2 2 5two 0 3 4 1 4 3 2 5 2 3 6 1

默认情况下，那些列会从DataFrame中移除，但也可以将其保留下来：

In [285]: frame.set_index(['c', 'd'], drop=False)Out[285]: a b c dc done 0 0 7 one 0 1 1 6 one 1 2 2 5 one 2two 0 3 4 two 0 1 4 3 two 1 2 5 2 two 2 3 6 1 two 3

reset_index的功能跟set_index刚好相反，层次化索引的级别会被转移到列里面：

In [286]: frame2.reset_index()Out[286]: c d a b0 one 0 0 71 one 1 1 62 one 2 2 53 two 0 3 44 two 1 4 35 two 2 5 26 two 3 6 1

其他有关pandas的话题

这里是另外一些可能在你的数据旅程中用得着的有关pandas的话题。

整数索引

操作由整数索引的pandas对象常常会让新手抓狂，因为它们跟内置的Python数据结构（如列表和元组）在索引语义上有些不同。例如，你可能认为下面这段代码不会产生一个错误：

ser = Series(np.arange(3.))ser[-1]

在这种情况下，虽然pandas会“求助于”整数索引，但没有哪种方法（至少我就不知道）能够既不引入任何bug又安全有效地解决该问题。这里，我们有一个含有0、1、2的索引，但是很难推断出用户想要什么（基于标签或位置的索引）：

In [288]: serOut[288]:0 01 12 2

相反，对于一个非整数索引，就没有这样的歧义：

In [289]: ser2 = Series(np.arange(3.), index=['a', 'b', 'c'])In [290]: ser2[-1]Out[290]: 2.0

为了保持良好的一致性，如果你的轴索引含有索引器，那么根据整数进行数据选取的操作将总是面向标签的。这也包括用ix进行切片：

In [291]: ser.ix[:1]Out[291]:0 01 1

如果你需要可靠的、不考虑索引类型的、基于位置的索引，可以使用Series 的iget_value方法和DataFrame的irow和icol方法：

In [292]: ser3 = Series(range(3), index=[-5, 1, 3])In [293]: ser3.iget_value(2)Out[293]: 2In [294]: frame = DataFrame(np.arange(6).reshape(3, 2), index=[2, 0, 1])In [295]: frame.irow(0)Out[295]:0 01 1Name: 2

面板数据

pandas有一个Panel数据结构（不是本书的主要内容），你可以将其看做一个三维版的DataFrame。pandas的大部分开发工作都集中在表格型数据的操作上，因为这些数据更常见，而且层次化索引也使得多数情况下没必要使用真正的N维数组。

你可以用一个由DataFrame对象组成的字典或一个三维ndarray来创建Panel对象：

import pandas.io.data as webpdata = pd.Panel(dict((stk, web.get_data_yahoo(stk, '1/1/2009', '6/1/2012')) for stk in ['AAPL', 'GOOG', 'MSFT', 'DELL']))

Panel中的每一项（类似于DataFrame的列）都是一个DataFrame：

In [297]: pdataOut[297]:Dimensions: 4 (items) x 861 (major) x 6 (minor)Items: AAPL to MSFTMajor axis: 2009-01-02 00:00:00 to 2012-06-01 00:00:00Minor axis: Open to Adj CloseIn [298]: pdata = pdata.swapaxes('items', 'minor')In [299]: pdata['Adj Close']Out[299]:DatetimeIndex: 861 entries, 2009-01-02 00:00:00 to 2012-06-01 00:00:00Data columns:AAPL 861 non-null valuesDELL 861 non-null valuesGOOG 861 non-null valuesMSFT 861 non-null valuesdtypes: float64(4)

基于ix的标签索引被推广到了三个维度，因此我们可以选取指定日期或日期范围的所有数据，如下所示：

In [300]: pdata.ix[:, '6/1/2012', :]Out[300]: Open High Low Close Volume Adj CloseAAPL 569.16 572.65 560.52 560.99 18606700 560.99DELL 12.15 12.30 12.05 12.07 19396700 12.07GOOG 571.79 572.65 568.35 570.98 3057900 570.98MSFT 28.76 28.96 28.44 28.45 56634300 28.45In [301]: pdata.ix['Adj Close', '5/22/2012':, :]Out[301]: AAPL DELL GOOG MSFTDate2012-05-22 556.97 15.08 600.80 29.762012-05-23 570.56 12.49 609.46 29.112012-05-24 565.32 12.45 603.66 29.072012-05-25 562.29 12.46 591.53 29.062012-05-29 572.27 12.66 594.34 29.562012-05-30 579.17 12.56 588.23 29.342012-05-31 577.73 12.33 580.86 29.192012-06-01 560.99 12.07 570.98 28.45

另一个用于呈现面板数据（尤其是对拟合统计模型）的办法是“堆积式的”DataFrame形式：

In [302]: stacked = pdata.ix[:, '5/30/2012':, :].to_frame()In [303]: stackedOut[303]: Open High Low Close Volume Adj Closemajor minor2012-05-30 AAPL 569.20 579.99 566.56 579.17 18908200 579.17 DELL 12.59 12.70 12.46 12.56 19787800 12.56 GOOG 588.16 591.90 583.53 588.23 1906700 588.23 MSFT 29.35 29.48 29.12 29.34 41585500 29.342012-05-31 AAPL 580.74 581.50 571.46 577.73 17559800 577.73 DELL 12.53 12.54 12.33 12.33 19955500 12.33 GOOG 588.72 590.00 579.00 580.86 2968300 580.86 MSFT 29.30 29.42 28.94 29.19 39134000 29.192012-06-01 AAPL 569.16 572.65 560.52 560.99 18606700 560.99 DELL 12.15 12.30 12.05 12.07 19396700 12.07 GOOG 571.79 572.65 568.35 570.98 3057900 570.98 MSFT 28.76 28.96 28.44 28.45 56634300 28.45

DataFrame有一个相应的to_panel方法，它是to_frame的逆运算：

In [304]: stacked.to_panel()Out[304]:Dimensions: 6 (items) x 3 (major) x 4 (minor)Items: Open to Adj CloseMajor axis: 2012-05-30 00:00:00 to 2012-06-01 00:00:00Minor axis: AAPL to MSFT

第6章　数据加载、存储与文件格式

如果不能将数据导入导出Python，本书所介绍的这些工具就没什么大用。我打算着重介绍pandas的输入输出对象，虽然别的库中也有不少以此为目的的工具。例如，NumPy提供了一个低级但异常高效的二进制数据加载和存储机制，包括对内存映射数组的支持等。详细内容请参阅第12章。

输入输出通常可以划分为几个大类：读取文本文件和其他更高效的磁盘存储格式，加载数据库中的数据，利用Web API操作网络资源。

读写文本格式的数据

因为其简单的文件交互语法、直观的数据结构，以及诸如元组打包解包之类的便利功能，Python在文本和文件处理方面已经成为一门招人喜欢的语言。

pandas提供了一些用于将表格型数据读取为DataFrame对象的函数。表6-1对它们进行了总结，其中read_csv和read_table可能会是你今后用得最多的。

表6-1：pandas中的解析函数

函数	说明
read_csv	从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为逗号
read_table	从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为制表符（“\t”）
read_fwf	读取定宽列格式数据（也就是说，没有分隔符）
read_clipboard	读取剪贴板中的数据，可以看做read_table的剪贴板版

我将大致介绍一下这些函数在将文本数据转换为DataFrame时所用到的一些技术。这些函数的选项可以划分为以下几个大类：

·索引：将一个或多个列当做返回的DataFrame处理，以及是否从文件、用户获取列名。

·类型推断和数据转换：包括用户定义值的转换、缺失值标记列表等。

·日期解析：包括组合功能，比如将分散在多个列中的日期时间信息组合成结果中的单个列。

·迭代：支持对大文件进行逐块迭代。

·不规整数据问题：跳过一些行、页脚、注释或其他一些不重要的东西（比如由成千上万个逗号隔开的数值数据）。

类型推断（type inference）是这些函数中最重要的功能之一，也就是说，你不需要指定列的类型到底是数值、整数、布尔值，还是字符串。日期和其他自定义类型的处理需要多花点工夫才行。首先我们来看一个以逗号分隔的（CSV）文本文件：

In [846]: !cat ch06/ex1.csv译注1a,b,c,d,message1,2,3,4,hello5,6,7,8,world9,10,11,12,foo

由于该文件以逗号分隔，所以我们可以使用read_csv将其读入一个DataFrame：

In [847]: df = pd.read_csv('ch06/ex1.csv')In [848]: dfOut[848]: a b c d message0 1 2 3 4 hello1 5 6 7 8 world2 9 10 11 12 foo

我们也可以用read_table，只不过需要指定分隔符而已：

In [849]: pd.read_table('ch06/ex1.csv', sep=',')Out[849]: a b c d message0 1 2 3 4 hello1 5 6 7 8 world2 9 10 11 12 foo

注意：这里我用的是cat这个UNIX shell命令将文本的原始内容打印到屏幕上。如果你用的是Windows，则可以使用type来达到同样的目的。

并不是所有文件都有标题行。看看下面这个文件：

In [850]: !cat ch06/ex2.csv1,2,3,4,hello5,6,7,8,world9,10,11,12,foo

读入该文件的办法有两个。你可以让pandas为其分配默认的列名，也可以自己定义列名：

In [851]: pd.read_csv('ch06/ex2.csv', header=None)Out[851]: X.1 X.2 X.3 X.4 X.50 1 2 3 4 hello1 5 6 7 8 world2 9 10 11 12 fooIn [852]: pd.read_csv('ch06/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])Out[852]: a b c d message0 1 2 3 4 hello1 5 6 7 8 world2 9 10 11 12 foo

假设你希望将message列做成DataFrame的索引。你可以明确表示要将该列放到索引4的位置上，也可以通过index_col参数指定"message"：

In [853]: names = ['a', 'b', 'c', 'd', 'message']In [854]: pd.read_csv('ch06/ex2.csv', names=names, index_col='message')Out[854]: a b c dmessagehello 1 2 3 4world 5 6 7 8foo 9 10 11 12

如果希望将多个列做成一个层次化索引，只需传入由列编号或列名组成的列表即可：

In [855]: !cat ch06/csv_mindex.csvkey1,key2,value1,value2one,a,1,2one,b,3,4one,c,5,6one,d,7,8two,a,9,10two,b,11,12two,c,13,14two,d,15,16In [856]: parsed = pd.read_csv('ch06/csv_mindex.csv', index_col=['key1', 'key2'])In [857]: parsedOut[857]: value1 value2key1 key2one a 1 2 b 3 4 c 5 6 d 7 8two a 9 10 b 11 12 c 13 14 d 15 16

有些表格可能不是用固定的分隔符去分隔字段的（比如空白符或其他模式译注2）。对于这种情况，可以编写一个正则表达式来作为read_table的分隔符。看看下面这个文本文件：

In [858]: list(open('ch06/ex3.txt'))Out[858]:[' A B C\n', 'aaa -0.264438 -1.026059 -0.619500\n', 'bbb 0.927272 0.302904 -0.032399\n', 'ccc -0.264273 -0.386314 -0.217601\n', 'ddd -0.871858 -0.348382 1.100491\n']

该文件各个字段由数量不定的空白符分隔，虽然你可以对其做一些手工调整，但这个情况还是处理比较好。本例的这个情况可以用正则表达式\s+表示，于是我们就有了：

In [859]: result = pd.read_table('ch06/ex3.txt', sep='\s+')In [860]: resultOut[860]: A B Caaa -0.264438 -1.026059 -0.619500bbb 0.927272 0.302904 -0.032399ccc -0.264273 -0.386314 -0.217601ddd -0.871858 -0.348382 1.100491

这里，由于列名比数据行的数量少译注3，所以read_table推断第一列应该是DataFrame的索引。

这些解析器函数还有许多参数可以帮助你处理各种各样的异形文件格式（参见表6-2）。比如说，你可以用skiprows跳过文件的第一行、第三行和第四行：

In [861]: !cat ch06/ex4.csv# hey!a,b,c,d,message# just wanted to make things more difficult for you# who reads CSV files with computers, anyway?1,2,3,4,hello5,6,7,8,world9,10,11,12,fooIn [862]: pd.read_csv('ch06/ex4.csv', skiprows=[0, 2, 3])Out[862]: a b c d message0 1 2 3 4 hello1 5 6 7 8 world2 9 10 11 12 foo

缺失值处理是文件解析任务中的一个重要组成部分。缺失数据经常是要么没有（空字符串），要么用某个标记值表示。默认情况下，pandas会用一组经常出现的标记值进行识别，如NA、-1.#IND以及NULL等：

In [863]: !cat ch06/ex5.csvsomething,a,b,c,d,messageone,1,2,3,4,NAtwo,5,6,,8,worldthree,9,10,11,12,fooIn [864]: result = pd.read_csv('ch06/ex5.csv')In [865]: resultOut[865]: something a b c d message0 one 1 2 3 4 NaN1 two 5 6 NaN 8 world2 three 9 10 11 12 fooIn [866]: pd.isnull(result)Out[866]: something a b c d message0 False False False False False True1 False False False True False False2 False False False False False False

na_values可以接受一组用于表示缺失值的字符串：

In [867]: result = pd.read_csv('ch06/ex5.csv', na_values=['NULL'])In [868]: resultOut[868]: something a b c d message0 one 1 2 3 4 NaN1 two 5 6 NaN 8 world2 three 9 10 11 12 foo

可以用一个字典为各列指定不同的NA标记值：

In [869]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}In [870]: pd.read_csv('ch06/ex5.csv', na_values=sentinels)Out[870]: something a b c d message0 one 1 2 3 4 NaN1 NaN 5 6 NaN 8 world2 three 9 10 11 12 NaN

表6-2：read_csv/read_table函数的参数

参数	说明
path	表示文件系统位置、URL、文件型对象的字符串
sep或delimiter	用于对行中各字段进行拆分的字符序列或正则表达式
header	用作列名的行号。默认为0（第一行），如果没有header行就应该设置为None
index_col	用作行索引的列编号或列名。可以是单个名称/数字或由多个名称/数字组成的列表（层次化索引）
names	用于结果的列名列表，结合header=None
skiprows	需要忽略的行数（从文件开始处算起），或需要跳过的行号列表（从0开始）
na_values	一组用于替换NA的值
comment	用于将注释信息从行尾拆分出去的字符（一个或多个）
parse_dates	尝试将数据解析为日期，默认为False。如果为True，则尝试解析所有列。此外，还可以指定需要解析的一组列号或列名。如果列表的元素为列表或元组，就会将多个列组合到一起再进行日期解析工作（例如，日期/时间分别位于两个列中）
keep_date_col	如果连接多列解析日期，则保持参与连接的列。默认为False。
converters	由列号/列名跟函数之间的映射关系组成的字典。例如，{'foo': f}会对foo列的所有值应用函数f
dayfirst	当解析有歧义的日期时，将其看做国际格式（例如，7/6/2012 → June 7, 2012）。默认为False
date_parser	用于解析日期的函数
nrows	需要读取的行数（从文件开始处算起）
iterator	返回一个TextParser以便逐块读取文件
chunksize	文件块的大小（用于迭代）
skip_footer	需要忽略的行数（从文件末尾处算起）

表6-2：read_csv/read_table函数的参数（续）

参数	说明
verbose	打印各种解析器输出信息，比如“非数值列中缺失值的数量”等
encoding	用于unicode的文本编码格式。例如，“utf-8”表示用UTF-8编码的文本
squeeze	如果数据经解析后仅含一列，则返回Series
thousands	千分位分隔符，如“,”或“.”

逐块读取文本文件

在处理很大的文件时，或找出大文件中的参数集以便于后续处理时，你可能只想读取文件的一小部分或逐块对文件进行迭代。

In [871]: result = pd.read_csv('ch06/ex6.csv')In [872]: resultOut[872]:Int64Index: 10000 entries, 0 to 9999Data columns:one 10000 non-null valuestwo 10000 non-null valuesthree 10000 non-null valuesfour 10000 non-null valueskey 10000 non-null valuesdtypes: float64(4), object(1)

如果只想读取几行（避免读取整个文件），通过nrows进行指定即可：

In [873]: pd.read_csv('ch06/ex6.csv', nrows=5)Out[873]: one two three four key0 0.467976 -0.038649 -0.295344 -1.824726 L1 -0.358893 1.404453 0.704965 -0.200638 B2 -0.501840 0.659254 -0.421691 -0.057688 G3 0.204886 1.074134 1.388361 -0.982404 R4 0.354628 -0.133116 0.283763 -0.837063 Q

要逐块读取文件，需要设置chunksize（行数）：

In [874]: chunker = pd.read_csv('ch06/ex6.csv', chunksize=1000)In [875]: chunkerOut[875]:

read_csv所返回的这个TextParser对象使你可以根据chunksize对文件进行逐块迭代。比如说，我们可以迭代处理ex6.csv，将值计数聚合到"key"列中，如下所示：

chunker = pd.read_csv('ch06/ex6.csv', chunksize=1000)tot = Series([])for piece in chunker: tot = tot.add(piece['key'].value_counts(), fill_value=0)tot = tot.order(ascending=False)

于是我们就有了：

In [877]: tot[:10]Out[877]:E 368X 364L 346O 343Q 340M 338J 337F 335K 334H 330

TextParser还有一个get_chunk方法，它使你可以读取任意大小的块。

将数据写出到文本格式

数据也可以被输出为分隔符格式的文本。我们再来看看之前读过的一个CSV文件：

In [878]: data = pd.read_csv('ch06/ex5.csv')In [879]: dataOut[879]: something a b c d message0 one 1 2 3 4 NaN1 two 5 6 NaN 8 world2 three 9 10 11 12 foo

利用DataFrame的to_csv方法，我们可以将数据写到一个以逗号分隔的文件中：

In [880]: data.to_csv('ch06/out.csv')In [881]: !cat ch06/out.csv,something,a,b,c,d,message0,one,1,2,3.0,4,1,two,5,6,,8,world2,three,9,10,11.0,12,foo

当然，还可以使用其他分隔符（由于这里直接写出到sys.stdout，所以仅仅是打印出文本结果而已）：

In [882]: data.to_csv(sys.stdout, sep='|')|something|a|b|c|d|message0|one|1|2|3.0|4|1|two|5|6||8|world2|three|9|10|11.0|12|foo

缺失值在输出结果中会被表示为空字符串。你可能希望将其表示为别的标记值：

In [883]: data.to_csv(sys.stdout, na_rep='NULL'),something,a,b,c,d,message0,one,1,2,3.0,4,NULL1,two,5,6,NULL,8,world2,three,9,10,11.0,12,foo

如果没有设置其他选项，则会写出行和列的标签。当然，它们也都可以被禁用：

In [884]: data.to_csv(sys.stdout, index=False, header=False)one,1,2,3.0,4,two,5,6,,8,worldthree,9,10,11.0,12,foo

此外，你还可以只写出一部分的列，并以你指定的顺序排列：

In [885]: data.to_csv(sys.stdout, index=False, cols=['a', 'b', 'c'])a,b,c1,2,3.05,6,9,10,11.0

Series也有一个to_csv方法：

In [886]: dates = pd.date_range('1/1/2000', periods=7)In [887]: ts = Series(np.arange(7), index=dates)In [888]: ts.to_csv('ch06/tseries.csv')In [889]: !cat ch06/tseries.csv2000-01-01 00:00:00,02000-01-02 00:00:00,12000-01-03 00:00:00,22000-01-04 00:00:00,32000-01-05 00:00:00,42000-01-06 00:00:00,52000-01-07 00:00:00,6

虽然只需一点整理工作（无header行，第一列作索引）就能用read_csv将CSV文件读取为Series，但还有一个更为方便的from_csv方法：

In [890]: Series.from_csv('ch06/tseries.csv', parse_dates=True)Out[890]:2000-01-01 02000-01-02 12000-01-03 22000-01-04 32000-01-05 42000-01-06 52000-01-07 6

更多信息请在IPython中查看to_csv和from_csv的文档。

手工处理分隔符格式

大部分存储在磁盘上的表格型数据都能用pandas.read_table进行加载。然而，有时还是需要做一些手工处理。由于接收到含有畸形行的文件而使read_table出毛病的情况并不少见。为了说明这些基本工具，看看下面这个简单的CSV文件：

In [891]: !cat ch06/ex7.csv"a","b","c""1","2","3""1","2","3","4"

对于任何单字符分隔符文件，可以直接使用Python内置的csv模块。将任意已打开的文件或文件型的对象传给csv.reader：

import csvf = open('ch06/ex7.csv')reader = csv.reader(f)

对这个reader进行迭代将会为每行产生一个元组译注4（并移除了所有的引号）：

In [893]: for line in reader: ....: print line['a', 'b', 'c']['1', '2', '3']['1', '2', '3', '4']

现在，为了使数据格式合乎要求，你需要对其做一些整理工作：

In [894]: lines = list(csv.reader(open('ch06/ex7.csv')))In [895]: header, values = lines[0], lines[1:]In [896]: data_dict = {h: v for h, v in zip(header, zip(*values))}In [897]: data_dictOut[897]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}

CSV文件的形式有很多。只需定义csv.Dialect的一个子类即可定义出新格式（如专门的分隔符、字符串引用约定、行结束符等）：

class my_dialect(csv.Dialect): lineterminator = '\n' delimiter = ';' quotechar = '"'reader = csv.reader(f, diaect=my_dialect)

各个CSV语支的参数也可以关键字的形式提供给csv.reader，而无需定义子类：

reader = csv.reader(f, delimiter='|')

可用的选项（csv.Dialect的属性）及其功能如表6-3所示。

表6-3：CSV语支选项

参数	说明
delimiter	用于分隔字段的单字符字符串。默认为“,”
lineterminator	用于写操作的行结束符，默认为“\r\n”。读操作将忽略此选项，它能认出跨平台的行结束符
quotechar	用于带有特殊字符（如分隔符）的字段的引用符号。默认为“"”
quoting	引用约定。可选值包括csv.QUOTE_ALL（引用所有字段）、csv.QUOTE_MINIMAL（只引用带有诸如分隔符之类特殊字符的字段）、csv.QUOTE_NONNUMERIC以及csv.QUOTE_NON（不引用）。完整信息请参考Python的文档。默认为QUOTE_MINIMAL
skipinitialspace	忽略分隔符后面的空白符。默认为False
doublequote	如何处理字段内的引用符号。如果为True，则双写。完整信息及行为请参见在线文档
escapechar	用于对分隔符进行转义的字符串（如果quoting被设置为csv.QUOTE_NONE的话）。默认禁用

注意：对于那些使用复杂分隔符或多字符分隔符的文件，csv模块就无能为力了。这种情况下，你就只能使用字符串的split方法或正则表达式方法re.split进行行拆分和其他整理工作了。

要手工输出分隔符文件，你可以使用csv.writer。它接受一个已打开且可写的文件对象以及跟csv.reader相同的那些语支和格式化选项：

with open('mydata.csv', 'w') as f:writer = csv.writer(f, dialect=my_dialect)writer.writerow(('one', 'two', 'three'))writer.writerow(('1', '2', '3'))writer.writerow(('4', '5', '6'))writer.writerow(('7', '8', '9'))

JSON数据

JSON（JavaScript Object Notation的简称）已经成为通过HTTP请求在Web浏览器和其他应用程序之间发送数据的标准格式之一。它是一种比表格型文本格式（如CSV）灵活得多的数据格式。下面是一个例子：

obj = """{"name": "Wes", "places_lived": ["United States", "Spain", "Germany"], "pet": null, "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"}, {"name": "Katie", "age": 33, "pet": "Cisco"}]}"""

除其空值null和一些其他的细微差别（如列表末尾不允许存在多余的逗号）之外，JSON非常接近于有效的Python代码。基本类型有对象（字典）、数组（列表）、字符串、数值、布尔值以及null。对象中所有的键都必须是字符串。许多Python库都可以读写JSON数据。我将使用json，因为它是构建于Python标准库中的。通过json.loads即可将JSON字符串转换成Python形式：

In [899]: import jsonIn [900]: result = json.loads(obj)In [901]: resultOut[901]:{u'name': u'Wes', u'pet': None, u'places_lived': [u'United States', u'Spain', u'Germany'], u'siblings': [{u'age': 25, u'name': u'Scott', u'pet': u'Zuko'}, {u'age': 33, u'name': u'Katie', u'pet': u'Cisco'}]}

相反，json.dumps则将Python对象转换成JSON格式：

In [902]: asjson = json.dumps(result)

如何将（一个或一组）JSON对象转换为DataFrame或其他便于分析的数据结构就由你决定了。最简单方便的方式是：向DataFrame构造器传入一组JSON对象，并选取数据字段的子集译注5。

In [903]: siblings = DataFrame(result['siblings'], columns=['name', 'age'])In [904]: siblingsOut[904]: name age0 Scott 251 Katie 33

第7章中关于USDA Food Database的那个例子进一步讲解了JSON数据的读取和处理（包括嵌套记录）。

注意：pandas团队正致力于为pandas添加原生的高效JSON导出（to_json）和解码（from_json）功能。不过目前还没开发完成。

XML和HTML：Web信息收集

Python有许多可以读写HTML和XML格式数据的库。lxml（http://lxml.de）就是其中之一，它能够高效且可靠地解析大文件。lxml有多个编程接口。首先我要用lxml.html处理HTML，然后再用lxml.objectify做一些XML处理。

许多网站都将数据放到HTML表格中以便在浏览器中查看，但不能以一种更易于机器阅读的格式（如JSON、HTML或XML）进行下载。我发现Yahoo!Finance的股票期权数据就是这样。可能你对这种数据不熟悉：期权是指使你有权从现在开始到未来某个时间（到期日）内以某个特定价格（执行价）买进（看涨期权）或卖出（看跌期权）某公司股票的衍生合约。人们的看涨和看跌期权交易有多种执行价和到期日，这些数据都可以在Yahoo!Finance的各种表格中找到。

首先，找到你希望获取数据的URL，利用urllib2将其打开，然后用lxml解析得到的数据流，如下所示：

from lxml.html import parsefrom urllib2 import urlopenparsed = parse(urlopen('http://finance.yahoo.com/q/op?s=AAPL+Options'))doc = parsed.getroot()

通过这个对象，你可以获取特定类型的所有HTML标签（tag），比如含有所需数据的table标签。给这个简单的例子加点启发性，假设你想得到该文档中所有的URL链接。HTML中的链接是a标签。使用文档根节点的findall方法以及一个XPath（对文档的“查询”的一种表示手段）：

In [906]: links = doc.findall('.//a')In [907]: links[15:20]Out[907]:[,,,,]

但这些是表示HTML元素的对象。要得到URL和链接文本，你必须使用各对象的get方法（针对URL）和text_content方法（针对显示文本）：

In [908]: lnk = links[28]In [909]: lnkOut[909]: In [910]: lnk.get('href')Out[910]: 'http://biz.yahoo.com/special.html'In [911]: lnk.text_content()Out[911]: 'Special Editions'

因此，编写下面这条列表推导式（list comprehension）即可获取文档中的全部URL：

In [912]: urls = [lnk.get('href') for lnk in doc.findall('.//a')]In [913]: urls[-10:]Out[913]: ['http://info.yahoo.com/privacy/us/yahoo/finance/details.html','http://info.yahoo.com/relevantads/','http://docs.yahoo.com/info/terms/','http://docs.yahoo.com/info/copyright/copyright.html','http://help.yahoo.com/l/us/yahoo/finance/forms_index.html','http://help.yahoo.com/l/us/yahoo/finance/quotes/fitadelay.html','http://help.yahoo.com/l/us/yahoo/finance/quotes/fitadelay.html','http://www.capitaliq.com','http://www.csidata.com','http://www.morningstar.com/']

现在，从文档中找出正确表格的办法就是反复试验了。有些网站会给目标表格加上一个id属性。我确定有两个分别放置看涨数据和看跌数据的表格：

tables = doc.findall('.//table')calls = tables[9]puts = tables[13]

每个表格都有一个标题行，然后才是数据行：

In [915]: rows = calls.findall('.//tr')

对于标题行和数据行，我们希望获取每个单元格内的文本。对于标题行，就是th单元格，而对于数据行，则是td单元格：

def _unpack(row, kind='td'): elts = row.findall('.//%s' % kind) return [val.text_content() for val in elts]

这样，我们就得到了：

In [917]: _unpack(rows[0], kind='th')Out[917]: ['Strike', 'Symbol', 'Last', 'Chg', 'Bid', 'Ask', 'Vol', 'Open Int']In [918]: _unpack(rows[1], kind='td')Out[918]: ['295.00','AAPL120818C00295000','310.40',' 0.00','289.80','290.80','1','169']

现在，把所有步骤结合起来，将数据转换为一个DataFrame。由于数值型数据仍然是字符串格式，所以我们希望将部分列（可能不是全部）转换为浮点数格式。虽然你可以手工实现该功能，但是pandas恰好就有一个TextParser类可用于自动类型转换（read_csv和其他解析函数其实在内部都用到了它）：

from pandas.io.parsers import TextParserdef parse_options_data(table): rows = table.findall('.//tr') header = _unpack(rows[0], kind='th') data = [_unpack(r) for r in rows[1:]] return TextParser(data, names=header).get_chunk()

最后，我对那两个lxml表格对象调用该解析函数并得到最终的DataFrame：

In [920]: call_data = parse_options_data(calls)In [921]: put_data = parse_options_data(puts)In [922]: call_data[:10]Out[922]:Strike Symbol Last Chg Bid Ask Vol Open Int0 295 AAPL120818C00295000 310.40 0.0 289.80 290.80 1 1691 300 AAPL120818C00300000 277.10 1.7 284.80 285.60 2 4782 305 AAPL120818C00305000 300.97 0.0 279.80 280.80 10 3163 310 AAPL120818C00310000 267.05 0.0 274.80 275.65 6 2394 315 AAPL120818C00315000 296.54 0.0 269.80 270.80 22 885 320 AAPL120818C00320000 291.63 0.0 264.80 265.80 96 1736 325 AAPL120818C00325000 261.34 0.0 259.80 260.80 N/A 1087 330 AAPL120818C00330000 230.25 0.0 254.80 255.80 N/A 218 335 AAPL120818C00335000 266.03 0.0 249.80 250.65 4 469 340 AAPL120818C00340000 272.58 0.0 244.80 245.80 4 30

利用lxml.objectify解析XML

XML（Extensible Markup Language）是另一种常见的支持分层、嵌套数据以及元数据的结构化数据格式。本书所使用的这些文件实际上来自于一个很大的XML文档。

之前，我介绍了lxml库及其lxml.html接口。这里我将介绍另一个用于操作XML数据的接口，即lxml.objectify。

纽约大都会运输署（Metropolitan Transportation Authority，MTA）发布了一些有关其公交和列车服务的数据资料（http://www.mta.info/developers/download.html）。这里，我们将看看包含在一组XML文件中的运行情况数据。每项列车或公交服务都有各自的文件（如Metro-

North Railroad的文件是Performance*MNR.xml[译注6](#809468440711498-YiZhu6_Gai_Wen_Jian_Yi_Jing_Geng_Ming__Dan_Huan_Shi_Ke_Yi_Xia_Zai_Dao_Xiang_Guan_De_Wen_Jian*)），其中每条XML记录就是一条月度数据，如下所示：

<INDICATOR>373889Metro-North RailroadEscalator Availability<DESCRIPTION>Percent of the time that escalators are operationalsystemwide. The availability rate is based on physical observations performed the morning of regular business days only. This is a new indicator the agency began reporting in 2009.</DESCRIPTION>201112<CATEGORY>Service Indicators</CATEGORY>
<FREQUENCY>M</FREQUENCY>U%197.0097.00</INDICATOR>

我们先用lxml.objectify解析该文件，然后通过getroot得到该XML文件的根节点的引用：

from lxml import objectifypath = 'Performance_MNR.xml'parsed = objectify.parse(open(path))root = parsed.getroot()

root.INDICATOR返回一个用于产生各个<INDICATOR>XML元素的生成器。对于每条记录，我们可以用标记名（如YTD*ACTUAL*）和数据值填充一个字典（排除几个标记）*[译注7](#809468440711498-YiZhu7You_Yu_Shu_Ju_Wen_Jian_Ge_Shi_Yi_Jing_Gai_Bian**Suo_Yi_Zhe_Duan_Dai_Ma_Bu_Neng_Zhi_Jie_Zhi_Xing_Le**Xu_Yao_An_Zhao_Xin_De_Shu_Ju_Ge_Shi_Shao_Wei_Diao_Zheng_Yi_Xia__Bu_Guo*)：

data = []skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ', 'DESIRED_CHANGE', 'DECIMAL_PLACES']for elt in root.INDICATOR: el_data = {} for child in elt.getchildren(): if child.tag in skip_fields: continue el_data[child.tag] = child.pyval data.append(el_data)

最后，将这组字典转换为一个DataFrame：

In [927]: perf = DataFrame(data)In [928]: perfOut[928]:Int64Index: 648 entries, 0 to 647Data columns:AGENCY_NAME 648 non-null valuesCATEGORY 648 non-null valuesDESCRIPTION 648 non-null valuesFREQUENCY 648 non-null valuesINDICATOR_NAME 648 non-null valuesINDICATOR_UNIT 648 non-null valuesMONTHLY_ACTUAL 648 non-null valuesMONTHLY_TARGET 648 non-null valuesPERIOD_MONTH 648 non-null valuesPERIOD_YEAR 648 non-null valuesYTD_ACTUAL 648 non-null valuesYTD_TARGET 648 non-null valuesdtypes: int64(2), object(10)Empty DataFrameColumns: array([], dtype=int64)Index: array([], dtype=int64)

XML数据可以比本例复杂得多。每个标记都可以有元数据。看看下面这个HTML的链接标记（它也算是一段有效的XML）：

from StringIO import StringIOtag = 'Google'root = objectify.parse(StringIO(tag)).getroot()

现在就可以访问链接文本或标记中的任何字段了（如href）：

In [930]: rootOut[930]: In [931]: root.get('href')Out[931]: 'http://www.google.com'In [932]: root.textOut[932]: 'Google'

译注1

：还是那句话，作者用的是UNIX，Windows下得用type。

译注2

：这里的“模式”一词表示的是“字符串”。如果对此概念较模糊，建议阅读《数据结构》。

译注3

：准确的说法应该是：列名的数量比列的数量少1。完整的说法应该是：列名“行”中“有内容的”字段数量比其他数据“行”中“有内容的”字段数量少1。

译注4

：很明显，这里得到的结果不是元组而是列表。

译注5

：意思是说可以选一部分字段。当然也可以全部选完。

译注6

：该文件已经更名，但还是可以下载到相关的文件。

译注7

：由于数据文件格式已经改变，所以这段代码不能直接执行了，需要按照新的数据格式稍微调整一下，不过也不麻烦，留给读者当做练习吧。

二进制数据格式

实现数据的二进制格式存储最简单的办法之一是使用Python内置的pickle序列化。为了使用方便，pandas对象都有一个用于将数据以pickle形式保存到磁盘上的save方法：

In [933]: frame = pd.read_csv('ch06/ex1.csv')In [934]: frameOut[934]: a b c d message0 1 2 3 4 hello1 5 6 7 8 world2 9 10 11 12 fooIn [935]: frame.save('ch06/frame_pickle')

你可以通过另一个也很好用的pickle函数pandas.load将数据读回到Python：

In [936]: pd.load('ch06/frame_pickle')Out[936]: a b c d message0 1 2 3 4 hello1 5 6 7 8 world2 9 10 11 12 foo

警告：pickle仅建议用于短期存储格式。其原因是很难保证该格式永远是稳定的；今天pickle的对象可能无法被后续版本的库unpickle出来。虽然我尽力保证这种事情不会发生在pandas中，但是今后的某个时候说不定还是得“打破”该pickle格式。

使用HDF5格式

很多工具都能实现高效读写磁盘上以二进制格式存储的科学数据。HDF5就是其中一个流行的工业级库，它是一个C库，带有许多语言的接口，如Java、Python和MATLAB等。HDF5中的HDF指的是层次型数据格式（hierarchical data format）。每个HDF5文件都含有一个文件系统式的节点结构，它使你能够存储多个数据集并支持元数据。与其他简单格式相比，HDF5支持多种压缩器的即时压缩，还能更高效地存储重复模式数据。对于那些非常大的无法直接放入内存的数据集，HDF5就是不错的选择，因为它可以高效地分块读写。

Python中的HDF5库有两个接口（即PyTables和h5py），它们各自采取了不同的问题解决方式。h5py提供了一种直接而高级的HDF5API访问接口，而PyTables则抽象了HDF5的许多细节以提供多种灵活的数据容器、表索引、查询功能以及对核外计算技术（out-of-core computation）的某些支持。

pandas有一个最小化的类似于字典的HDFStore类，它通过PyTables存储pandas对象：

In [937]: store = pd.HDFStore('mydata.h5')In [938]: store['obj1'] = frameIn [939]: store['obj1_col'] = frame['a']In [940]: storeOut[940]:File path: mydata.h5obj1 DataFrameobj1_col Series

HDF5文件中的对象可以通过与字典一样的方式进行获取：

In [941]: store['obj1']Out[941]: a b c d message0 1 2 3 4 hello1 5 6 7 8 world2 9 10 11 12 foo

如果需要处理海量数据，我建议你好好研究一下PyTables和h5py，看看它们能满足你的哪些需求。由于许多数据分析问题都是IO密集型（而不是CPU密集型），利用HDF5这样的工具能显著提升应用程序的效率。

警告： HDF5不是数据库。它最适合用作“一次写多次读”的数据集。虽然数据可以在任何时候被添加到文件中，但如果同时发生多个写操作，文件就可能会被破坏。

读取Microsoft Excel文件

pandas的ExcelFile类支持读取存储在Excel 2003（或更高版本）中的表格型数据。由于ExcelFile用到了xlrd和openpyxl包，所以你先得安装它们才行。通过传入一个xls或xlsx文件的路径即可创建一个ExcelFile实例：

xls_file = pd.ExcelFile('data.xls')

存放在某个工作表中的数据可以通过parse读取到DataFrame中：

table = xls_file.parse('Sheet1')

使用HTML和Web API

许多网站都有一些通过JSON或其他格式提供数据的公共API。通过Python访问这些API的办法有不少。一个简单易用的办法（推荐）是requests包（http://docs.python-requests.org）。为了在Twitter上搜索"python pandas"，我们可以发送一个HTTP GET请求，如下所示：

In [944]: import requestsIn [945]: url = 'http://search.twitter.com/search.json?q=python%20pandas'In [946]: resp = requests.get(url)In [947]: respOut[947]:

Response对象的text属性含有GET请求的内容。许多Web API返回的都是JSON字符串，我们必须将其加载到一个Python对象中：

In [948]: import jsonIn [949]: data = json.loads(resp.text)In [950]: data.keys()Out[950]: [u'next_page',u'completed_in',u'max_id_str',u'since_id_str',u'refresh_url',u'results',u'since_id',u'results_per_page',u'query',u'max_id',u'page']

响应结果中的results字段含有一组tweet，每条tweet被表示为一个Python字典，如下所示：

{u'created_at': u'Mon, 25 Jun 2012 17:50:33 +0000',u'from_user': u'wesmckinn',u'from_user_id': 115494880,u'from_user_id_str': u'115494880',u'from_user_name': u'Wes McKinney',u'geo': None,u'id': 217313849177686018,u'id_str': u'217313849177686018',u'iso_language_code': u'pt',u'metadata': {u'result_type': u'recent'},u'source': u'web',u'text': u'Lunchtime pandas-fu http://t.co/SI70xZZQ #pydata',u'to_user': None,u'to_user_id': 0,u'to_user_id_str': u'0',u'to_user_name': None}

我们用一个列表定义出感兴趣的tweet字段，然后将results列表传给DataFrame：

In [951]: tweet_fields = ['created_at', 'from_user', 'id', 'text']In [952]: tweets = DataFrame(data['results'], columns=tweet_fields)In [953]: tweetsOut[953]:Int64Index: 15 entries, 0 to 14Data columns:created_at 15 non-null valuesfrom_user 15 non-null valuesid 15 non-null valuestext 15 non-null valuesdtypes: int64(1), object(3)

现在，DataFrame中的每一行就有了来自一条tweet的数据：

In [121]: tweets.ix[7]Out[121]:created_at Thu, 23 Jul 2012 09:54:00 +0000from_user deblikeid 227419585803059201text pandas: powerful Python data analysis toolkitName: 7

要想能够直接得到便于分析的DataFrame对象，只需再多费些精力创建出对常见Web API的更高级接口即可。

使用数据库

在许多应用中，数据很少取自文本文件，因为用这种方式存储大量数据很低效。基于SQL的关系型数据库（如SQL Server、PostgreSQL和MySQL等）使用非常广泛，此外还有一些非SQL（即所谓的NoSQL）型数据库也变得非常流行。数据库的选择通常取决于性能、数据完整性以及应用程序的伸缩性需求。

将数据从SQL加载到DataFrame的过程很简单，此外pandas还有一些能够简化该过程的函数。例如，我将使用一款嵌入式的SQLite数据库（通过Python内置的sqlite3驱动器）：

import sqlite3query = """CREATE TABLE test(a VARCHAR(20), b VARCHAR(20),c REAL, d INTEGER);"""con = sqlite3.connect(':memory:')con.execute(query)con.commit()

然后插入几行数据：

data = [('Atlanta', 'Georgia', 1.25, 6), ('Tallahassee', 'Florida', 2.6, 3), ('Sacramento', 'California', 1.7, 5)]stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"con.executemany(stmt, data)con.commit()

从表中选取数据时，大部分Python SQL驱动器（PyODBC、psycopg2、MySQLdb、pymssql等）都会返回一个元组列表：

In [956]: cursor = con.execute('select * from test')In [957]: rows = cursor.fetchall()In [958]: rowsOut[958]:[(u'Atlanta', u'Georgia', 1.25, 6), (u'Tallahassee', u'Florida', 2.6, 3), (u'Sacramento', u'California', 1.7, 5)]

你可以将这个元组列表传给DataFrame的构造器，但还需要列名（位于游标的description属性中）：

In [959]: cursor.descriptionOut[959]:(('a', None, None, None, None, None, None), ('b', None, None, None, None, None, None), ('c', None, None, None, None, None, None), ('d', None, None, None, None, None, None))In [960]: DataFrame(rows, columns=zip(*cursor.description)[0])Out[960]: a b c d0 Atlanta Georgia 1.25 61 Tallahassee Florida 2.60 32 Sacramento California 1.70 5

这种数据规整操作相当多，你肯定不想每查一次数据库就重写一次。pandas有一个可以简化该过程的read_frame函数（位于pandas.io.sql模块）。只需传入select语句和连接对象即可：

In [961]: import pandas.io.sql as sqlIn [962]: sql.read_frame('select * from test', con)Out[962]: a b c d0 Atlanta Georgia 1.25 61 Tallahassee Florida 2.60 32 Sacramento California 1.70 5

存取MongoDB中的数据

NoSQL数据库有许多不同的形式。有些是简单的字典式键值对存储（如BerkeleyDB和Tokyo Cabinet），另一些则是基于文档的（其中的基本单元是字典型的对象）。本例选用的是MongoDB（http://mongodb.org）。我先在自己的电脑上启动一个MongoDB实例，然后用pymongo（MongoDB的官方驱动器）通过默认端口进行连接：

import pymongocon = pymongo.Connection('localhost', port=27017)

存储在MongoDB中的文档被组织在数据库的集合（collection）译注8中。MongoDB服务器的每个运行实例可以有多个数据库，而每个数据库又可以有多个集合。假设你想保存之前通过Twitter API获取的数据。首先，我可以访问tweets集合（暂时还是空的）：

tweets = con.db.tweets

然后，我将那组tweet加载进来并通过tweets.save（用于将Python字典写入MongoDB）逐个存入集合中：

import requests, jsonurl = 'http://search.twitter.com/search.json?q=python%20pandas'data = json.loads(requests.get(url).text)for tweet in data['results']: tweets.save(tweet)

现在，如果我想从该集合中取出我自己发的tweet（如果有的话），可以用下面的代码对集合进行查询：

cursor = tweets.find({'from_user': 'wesmckinn'})

返回的游标是一个迭代器，它可以为每个文档产生一个字典。跟之前一样，我可以将其转换为一个DataFrame。此外，还可以只获取各tweet的部分字段：

tweet_fields = ['created_at', 'from_user', 'id', 'text']result = DataFrame(list(cursor), columns=tweet_fields)

译注8

：如果实在不明白，可直接想象成表。

第7章　数据规整化：清理、转换、合并、重塑

数据分析和建模方面的大量编程工作都是用在数据准备上的：加载、清理、转换以及重塑。有时候，存放在文件或数据库中的数据并不能满足你的数据处理应用的要求。许多人都选择使用通用编程语言（如Python、Perl、R或Java）或UNIX文本处理工具（如sed或awk）对数据格式进行专门处理。幸运的是，pandas和Python标准库提供了一组高级的、灵活的、高效的核心函数和算法，它们使你能够轻松地将数据规整化为正确的形式。

如果你发现了一种本书或pandas库中没有的数据操作方式，请尽管在邮件列表或GitHub网站上提出。实际上，pandas的许多设计和实现都是由真实应用的需求所驱动的。

合并数据集

pandas对象中的数据可以通过一些内置的方式进行合并：

·pandas.merge可根据一个或多个键将不同DataFrame中的行连接起来。SQL或其他关系型数据库的用户对此应该会比较熟悉，因为它实现的就是数据库的连接操作。

·pandas.concat可以沿着一条轴将多个对象堆叠到一起。

·实例方法combine*first*可以将重复数据编接在一起，用一个对象中的值填充另一个对象中的缺失值。*[译注1](#809468440711498-YiZhu1Tong_Su_Lai_Shuo**Cha_Bu_Duo_Jiu_Shi_Shu_Ju_Ku_De_Quan_Wai_Lian_Jie**Zhu_Yi**Cha_Bu_Duo**He**Quan_Wai_Lian_Jie**Zhe_Liang_Ge_Ci__**Jian_Dan_Di_Shuo**Jiu_Shi*
)

我将分别对它们进行讲解，并给出一些例子。本书剩余部分的示例中将经常用到它们。

数据库风格的DataFrame合并

数据集的合并（merge）或连接（join）运算是通过一个或多个键将行链接起来的。这些运算是关系型数据库的核心。pandas的merge函数是对数据应用这些算法的主要切入点。

我们以一个简单的例子开始：

In [15]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], ...: 'data1': range(7)})In [16]: df2 = DataFrame({'key': ['a', 'b', 'd'], ...: 'data2': range(3)})In [17]: df1Out[17]: data1 key0 0 b1 1 b2 2 a3 3 c4 4 a5 5 a6 6 bIn [18]: df2Out[18]: data2 key0 0 a1 1 b2 2 d

这是一种多对一的合并。df1中的数据有多个被标记为a和b的行，而df2中key列的每个值则仅对应一行。对这些对象调用merge即可得到：

In [19]: pd.merge(df1, df2)Out[19]: data1 key data20 2 a 01 4 a 02 5 a 03 0 b 14 1 b 15 6 b 1

注意，我并没有指明要用哪个列进行连接。如果没有指定，merge就会将重叠列的列名当做键。不过，最好显式指定一下：

In [20]: pd.merge(df1, df2, on='key')Out[20]: data1 key data20 2 a 01 4 a 02 5 a 03 0 b 14 1 b 15 6 b 1

如果两个对象的列名不同，也可以分别进行指定：

In [21]: df3 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], ...: 'data1': range(7)})In [22]: df4 = DataFrame({'rkey': ['a', 'b', 'd'], ...: 'data2': range(3)})In [23]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')Out[23]: data1 lkey data2 rkey0 2 a 0 a1 4 a 0 a2 5 a 0 a3 0 b 1 b4 1 b 1 b5 6 b 1 b

可能你已经注意到了，结果里面c和d以及与之相关的数据消失了。默认情况下，merge做的是"inner"连接；结果中的键是交集。其他方式还有"left"、"right"以及"outer"。外连接求取的是键的并集，组合了左连接和右连接的效果：

In [24]: pd.merge(df1, df2, how='outer')Out[24]: data1 key data20 2 a 01 4 a 02 5 a 03 0 b 14 1 b 15 6 b 16 3 c NaN7 NaN d 2

多对多的合并操作非常简单，无需额外的工作。如下所示：

In [25]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'], ...: 'data1': range(6)})In [26]: df2 = DataFrame({'key': ['a', 'b', 'a', 'b', 'd'], ...: 'data2': range(5)})In [27]: df1Out[27]: data1 key0 0 b1 1 b2 2 a3 3 c4 4 a5 5 bIn [28]: df2Out[28]: data2 key0 0 a1 1 b2 2 a3 3 b4 4 dIn [29]: pd.merge(df1, df2, on='key', how='left')Out[29]: data1 key data20 2 a 01 2 a 22 4 a 03 4 a 24 0 b 15 0 b 36 1 b 17 1 b 38 5 b 19 5 b 310 3 c NaN

多对多连接产生的是行的笛卡尔积。由于左边的DataFrame有3个"b"行，右边的有2个，所以最终结果中就有6个"b"行。连接方式只影响出现在结果中的键：

In [30]: pd.merge(df1, df2, how='inner')Out[30]: data1 key data20 2 a 01 2 a 22 4 a 03 4 a 24 0 b 15 0 b 36 1 b 17 1 b 38 5 b 19 5 b 3

要根据多个键进行合并，传入一个由列名组成的列表即可：

In [31]: left = DataFrame({'key1': ['foo', 'foo', 'bar'], ...: 'key2': ['one', 'two', 'one'], ...: 'lval': [1, 2, 3]})In [32]: right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'], ...: 'key2': ['one', 'one', 'one', 'two'], ...: 'rval': [4, 5, 6, 7]})In [33]: pd.merge(left, right, on=['key1', 'key2'], how='outer')Out[33]: key1 key2 lval rval0 bar one 3 61 bar two NaN 72 foo one 1 43 foo one 1 54 foo two 2 NaN

结果中会出现哪些键组合取决于所选的合并方式，你可以这样来理解：多个键形成一系列元组，并将其当做单个连接键（当然，实际上并不是这么回事）。

警告： 在进行列－列连接时，DataFrame对象中的索引会被丢弃。

对于合并运算需要考虑的最后一个问题是对重复列名的处理。虽然你可以手工处理列名重叠的问题（稍后将会介绍如何重命名轴标签），但merge有一个更实用的suffixes选项，用于指定附加到左右两个DataFrame对象的重叠列名上的字符串：

In [34]: pd.merge(left, right, on='key1')Out[34]: key1 key2_x lval key2_y rval0 bar one 3 one 61 bar one 3 two 72 foo one 1 one 43 foo one 1 one 54 foo two 2 one 45 foo two 2 one 5In [35]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))Out[35]: key1 key2_left lval key2_right rval0 bar one 3 one 61 bar one 3 two 72 foo one 1 one 43 foo one 1 one 54 foo two 2 one 45 foo two 2 one 5

merge的参数请参见表7-1。索引上的连接将在下一节中讲解。

表7-1：merge函数的参数

参数	说明
left	参与合并的左侧DataFrame
right	参与合并的右侧DataFrame
how	“inner”、“outer”、“left”、“right”其中之一。默认为“inner”

表7-1：merge函数的参数（续）

参数	说明
on	用于连接的列名。必须存在于左右两个DataFrame对象中。如果未指定，且其他连接键也未指定，则以left和right列名的交集作为连接键
left_on	左侧DataFrame中用作连接键的列
right_on	右侧DataFrame中用作连接键的列
left_index	将左侧的行索引用作其连接键
right_index	类似于left_index
sort	根据连接键对合并后的数据进行排序，默认为True。有时在处理大数据集时，禁用该选项可获得更好的性能
suffixes	字符串值元组，用于追加到重叠列名的末尾，默认为('_x', '_y')。例如，如果左右两个DataFrame对象都有“data”，则结果中就会出现“data_x”和“data_y”
copy	设置为False，可以在某些特殊情况下避免将数据复制到结果数据结构中。默认总是复制

索引上的合并

有时候，DataFrame中的连接键位于其索引中。在这种情况下，你可以传入left_index=True或right_index=True（或两个都传）以说明索引应该被用作连接键：

In [36]: left1 = DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'], ...: 'value': range(6)})In [37]: right1 = DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])In [38]: left1Out[38]: key value0 a 01 b 12 a 23 a 34 b 45 c 5In [39]: right1Out[39]: group_vala 3.5b 7.0In [40]: pd.merge(left1, right1, left_on='key', right_index=True)Out[40]: key value group_val0 a 0 3.52 a 2 3.53 a 3 3.51 b 1 7.04 b 4 7.0

由于默认的merge方法是求取连接键的交集，因此你可以通过外连接的方式得到它们的并集：

In [41]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')Out[41]: key value group_val0 a 0 3.52 a 2 3.53 a 3 3.51 b 1 7.04 b 4 7.05 c 5 NaN

对于层次化索引的数据，事情就有点复杂了：

In [42]: lefth = DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'], ...: 'key2': [2000, 2001, 2002, 2001, 2002], ...: 'data': np.arange(5.)})In [43]: righth = DataFrame(np.arange(12).reshape((6, 2)), ...: index=[['Nevada', 'Nevada', 'Ohio', 'Ohio', 'Ohio', 'Ohio'], ...: [2001, 2000, 2000, 2000, 2001, 2002]], ...: columns=['event1', 'event2'])In [44]: lefthOut[44]: data key1 key20 0 Ohio 20001 1 Ohio 20012 2 Ohio 20023 3 Nevada 20014 4 Nevada 2002In [45]: righthOut[45]: event1 event2Nevada 2001 0 1 2000 2 3Ohio 2000 4 5 2000 6 7 2001 8 9 2002 10 11

这种情况下，你必须以列表的形式指明用作合并键的多个列（注意对重复索引值的处理）：

In [46]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)Out[46]: data key1 key2 event1 event23 3 Nevada 2001 0 10 0 Ohio 2000 4 50 0 Ohio 2000 6 71 1 Ohio 2001 8 92 2 Ohio 2002 10 11In [47]: pd.merge(lefth, righth, left_on=['key1', 'key2'], ...: right_index=True, how='outer')Out[47]: data key1 key2 event1 event24 NaN Nevada 2000 2 33 3 Nevada 2001 0 14 4 Nevada 2002 NaN NaN0 0 Ohio 2000 4 50 0 Ohio 2000 6 71 1 Ohio 2001 8 92 2 Ohio 2002 10 11

同时使用合并双方的索引也没问题：

In [48]: left2 = DataFrame([[1., 2.], [3., 4.], [5., 6.]], index=['a', 'c', 'e'], ...: columns=['Ohio', 'Nevada'])In [49]: right2 = DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]], ...: index=['b', 'c', 'd', 'e'], columns=['Missouri', 'Alabama'])In [50]: left2Out[50]: Ohio Nevadaa 1 2c 3 4e 5 6In [51]: right2Out[51]: Missouri Alabamab 7 8c 9 10d 11 12e 13 14In [52]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)Out[52]: Ohio Nevada Missouri Alabamaa 1 2 NaN NaNb NaN NaN 7 8c 3 4 9 10d NaN NaN 11 12e 5 6 13 14

DataFrame还有一个join实例方法，它能更为方便地实现按索引合并。它还可用于合并多个带有相同或相似索引的DataFrame对象，而不管它们之间有没有重叠的列。在上面那个例子中，我们可以编写：

In [53]: left2.join(right2, how='outer')Out[53]: Ohio Nevada Missouri Alabamaa 1 2 NaN NaNb NaN NaN 7 8c 3 4 9 10d NaN NaN 11 12e 5 6 13 14

由于一些历史原因（早期版本的pandas），DataFrame的join方法是在连接键上做左连接。它还支持参数DataFrame的索引跟调用者DataFrame的某个列之间的连接：

In [54]: left1.join(right1, on='key')Out[54]: key value group_val0 a 0 3.51 b 1 7.02 a 2 3.53 a 3 3.54 b 4 7.05 c 5 NaN

最后，对于简单的索引合并，你还可以向join传入一组DataFrame（后面我们会介绍更为通用的concat函数，它也能实现此功能）：

In [55]: another = DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]], ...: index=['a', 'c', 'e', 'f'], columns=['New York', 'Oregon'])In [56]: left2.join([right2, another])Out[56]: Ohio Nevada Missouri Alabama New York Oregona 1 2 NaN NaN 7 8c 3 4 9 10 9 10e 5 6 13 14 11 12In [57]: left2.join([right2, another], how='outer')Out[57]: Ohio Nevada Missouri Alabama New York Oregona 1 2 NaN NaN 7 8b NaN NaN 7 8 NaN NaNc 3 4 9 10 9 10d NaN NaN 11 12 NaN NaNe 5 6 13 14 11 12f NaN NaN NaN NaN 16 17

轴向连接

另一种数据合并运算也被称作连接（concatenation）、绑定（binding）或堆叠（stacking）。NumPy有一个用于合并原始NumPy数组的concatenation函数：

In [58]: arr = np.arange(12).reshape((3, 4))In [59]: arrOut[59]:array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]])In [60]: np.concatenate([arr, arr], axis=1)Out[60]:array([[0, 1, 2, 3, 0, 1, 2, 3], [4, 5, 6, 7, 4, 5, 6, 7], [8, 9, 10, 11, 8, 9, 10, 11]])

对于pandas对象（如Series和DataFrame），带有标签的轴使你能够进一步推广数组的连接运算。具体点说，你还需要考虑以下这些东西：

·如果各对象其他轴上的索引不同，那些轴应该是做并集还是交集？

·结果对象中的分组需要各不相同吗？

·用于连接的轴重要吗？

pandas的concat函数提供了一种能够解决这些问题的可靠方式。我将给出一些例子来讲解其使用方式。假设有三个没有重叠索引的Series：

In [61]: s1 = Series([0, 1], index=['a', 'b'])In [62]: s2 = Series([2, 3, 4], index=['c', 'd', 'e'])In [63]: s3 = Series([5, 6], index=['f', 'g'])

对这些对象调用concat可以将值和索引粘合在一起：

In [64]: pd.concat([s1, s2, s3])Out[64]:a 0b 1c 2d 3e 4f 5g 6

默认情况下，concat是在axis=0上工作的，最终产生一个新的Series。如果传入axis=1，则结果就会变成一个DataFrame（axis=1是列）：

In [65]: pd.concat([s1, s2, s3], axis=1)Out[65]: 0 1 2a 0 NaN NaNb 1 NaN NaNc NaN 2 NaNd NaN 3 NaNe NaN 4 NaNf NaN NaN 5g NaN NaN 6

这种情况下，另外一条轴上没有重叠，从索引的有序并集（外连接）上就可以看出来。传入join='inner'即可得到它们的交集：

In [66]: s4 = pd.concat([s1 * 5, s3])In [67]: pd.concat([s1, s4], axis=1) In [68]: pd.concat([s1, s4], axis=1, join='inner')Out[67]: Out[68]: 0 1 0 1a 0 0 a 0 0b 1 5 b 1 5f NaN 5g NaN 6

你可以通过join_axes指定要在其他轴上使用的索引：

In [69]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])Out[69]: 0 1a 0 0c NaN NaNb 1 5e NaN NaN

不过有个问题，参与连接的片段在结果中区分不开。假设你想要在连接轴上创建一个层次化索引。使用keys参数即可达到这个目的：

In [70]: result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])In [71]: resultOut[71]:one a 0 b 1two a 0 b 1three f 5 g 6# 稍后将详细讲解unstack函数In [72]: result.unstack()Out[72]: a b f gone 0 1 NaN NaNtwo 0 1 NaN NaNthree NaN NaN 5 6

如果沿着axis=1对Series进行合并，则keys就会成为DataFrame的列头：

In [73]: pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])Out[73]: one two threea 0 NaN NaNb 1 NaN NaNc NaN 2 NaNd NaN 3 NaNe NaN 4 NaNf NaN NaN 5g NaN NaN 6

同样的逻辑对DataFrame对象也是一样：

In [74]: df1 = DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'], ...: columns=['one', 'two'])In [75]: df2 = DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'], ...: columns=['three', 'four'])In [76]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])Out[76]: level1 level2 one two three foura 0 1 5 6b 2 3 NaN NaNc 4 5 7 8

如果传入的不是列表而是一个字典，则字典的键就会被当做keys选项的值：

In [77]: pd.concat({'level1': df1, 'level2': df2}, axis=1)Out[77]: level1 level2 one two three foura 0 1 5 6b 2 3 NaN NaNc 4 5 7 8

此外还有两个用于管理层次化索引创建方式的参数（参见表7-2）：

In [78]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'], ...: names=['upper', 'lower'])Out[78]:upper level1 level2lower one two three foura 0 1 5 6b 2 3 NaN NaNc 4 5 7 8

最后一个需要考虑的问题是，跟当前分析工作无关的DataFrame行索引译注2：

In [79]: df1 = DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])In [80]: df2 = DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])In [81]: df1Out[81]: a b c d0 -0.204708 0.478943 -0.519439 -0.5557301 1.965781 1.393406 0.092908 0.2817462 0.769023 1.246435 1.007189 -1.296221In [82]: df2Out[82]: b d a0 0.274992 0.228913 1.3529171 0.886429 -2.001637 -0.371843

在这种情况下，传入ignore_index=True即可：

In [83]: pd.concat([df1, df2], ignore_index=True)Out[83]: a b c d0 -0.204708 0.478943 -0.519439 -0.5557301 1.965781 1.393406 0.092908 0.2817462 0.769023 1.246435 1.007189 -1.2962213 1.352917 0.274992 NaN 0.2289134 -0.371843 0.886429 NaN -2.001637

表7-2：concat函数的参数

参数	说明
objs	参与连接的pandas对象的列表或字典。唯一必需的参数
axis	指明连接的轴向，默认为0
join	“inner”、“outer”其中之一，默认为“outer”。指明其他轴向上的索引是按交集（inner）还是并集（outer）进行合并
join_axes	指明用于其他n-1条轴的索引，不执行并集/交集运算
keys	与连接对象有关的值，用于形成连接轴向上的层次化索引。可以是任意值的列表或数组、元组数组、数组列表（如果将levels设置成多级数组的话）
levels	指定用作层次化索引各级别上的索引，如果设置了keys的话[译注3]
names	用于创建分层级别的名称，如果设置了keys和（或）levels的话
verify_integrity	检查结果对象新轴上的重复情况，如果发现则引发异常。默认（False）允许重复
ignore_index	不保留连接轴上的索引，产生一组新索引range(total_length)

译注3：就是外层级别的索引。

合并重叠数据

还有一种数据组合问题不能用简单的合并（merge）或连接（concatenation）运算来处理。比如说，你可能有索引全部或部分重叠的两个数据集。给这个例子增加一点启发性，我们使用NumPy的where函数，它用于表达一种矢量化的if-else：

In [84]: a = Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan], ...: index=['f', 'e', 'd', 'c', 'b', 'a'])In [85]: b = Series(np.arange(len(a), dtype=np.float64), ...: index=['f', 'e', 'd', 'c', 'b', 'a'])In [86]: b[-1] = np.nanIn [87]: a In [88]: b In [89]: np.where(pd.isnull(a), b, a)Out[87]: Out[88]: Out[89]:f NaN f 0 f 0.0e 2.5 e 1 e 2.5d NaN d 2 d 2.0c 3.5 c 3 c 3.5b 4.5 b 4 b 4.5a NaN a NaN a NaN

Series有一个combine_first方法，实现的也是一样的功能，而且会进行数据对齐：

In [90]: b[:-2].combine_first(a[2:])Out[90]:a NaNb 4.5c 3.0d 2.0e 1.0f 0.0

对于DataFrame，combine_first自然也会在列上做同样的事情，因此你可以将其看做：用参数对象中的数据为调用者对象的缺失数据“打补丁”：

In [91]: df1 = DataFrame({'a': [1., np.nan, 5., np.nan], ...: 'b': [np.nan, 2., np.nan, 6.], ...: 'c': range(2, 18, 4)})In [92]: df2 = DataFrame({'a': [5., 4., np.nan, 3., 7.], ...: 'b': [np.nan, 3., 4., 6., 8.]})In [93]: df1.combine_first(df2)Out[93]: a b c0 1 NaN 21 4 2 62 5 4 103 3 6 144 7 8 NaN

译注1

：通俗来说，差不多就是数据库的全外连接（注意“差不多”和“全外连接”这两个词）。简单地说，就是先从第一个对象中选值，不行就再去第二个对象中选值。

译注2

：也就是说那些行索引是无意义的。

重塑和轴向旋转

有许多用于重新排列表格型数据的基础运算。这些函数也称作重塑（reshape）或轴向旋转（pivot）运算。

重塑层次化索引

层次化索引为DataFrame数据的重排任务提供了一种具有良好一致性的方式。主要功能有二：

·stack：将数据的列“旋转”为行。

·unstack：将数据的行“旋转”为列。

我将通过一系列的范例来讲解这些操作。接下来看一个简单的DataFrame，其中的行列索引均为字符串：

In [94]: data = DataFrame(np.arange(6).reshape((2, 3)), ...: index=pd.Index(['Ohio', 'Colorado'], name='state'), ...: columns=pd.Index(['one', 'two', 'three'], name='number'))In [95]: dataOut[95]:number one two threestateOhio 0 1 2Colorado 3 4 5

使用该数据的stack方法即可将列转换为行，得到一个Series：

In [96]: result = data.stack()In [97]: resultOut[97]:state numberOhio one 0 two 1 three 2Colorado one 3 two 4 three 5

对于一个层次化索引的Series，你可以用unstack将其重排为一个DataFrame：

In [98]: result.unstack()Out[98]:number one two threestateOhio 0 1 2Colorado 3 4 5

默认情况下，unstack操作的是最内层（stack也是如此）。传入分层级别的编号或名称即可对其他级别进行unstack操作：

In [99]: result.unstack(0) In [100]: result.unstack('state')Out[99]: Out[100]:state Ohio Colorado state Ohio Coloradonumber numberone 0 3 one 0 3two 1 4 two 1 4three 2 5 three 2 5

如果不是所有的级别值都能在各分组中找到的话，则unstack操作可能会引入缺失数据：

In [101]: s1 = Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])In [102]: s2 = Series([4, 5, 6], index=['c', 'd', 'e'])In [103]: data2 = pd.concat([s1, s2], keys=['one', 'two'])In [104]: data2.unstack()Out[104]: a b c d eone 0 1 2 3 NaNtwo NaN NaN 4 5 6

stack默认会滤除缺失数据，因此该运算是可逆的：

In [105]: data2.unstack().stack() In [106]: data2.unstack().stack(dropna=False) Out[105]: Out[106]:one a 0 one a 0 b 1 b 1 c 2 c 2 d 3 d 3two c 4 e NaN d 5 two a NaN e 6 b NaN c 4 d 5 e 6

在对DataFrame进行unstack操作时，作为旋转轴的级别将会成为结果中的最低级别：

In [107]: df = DataFrame({'left': result, 'right': result + 5}, ...: columns=pd.Index(['left', 'right'], name='side'))In [108]: dfOut[108]:side left rightstate numberOhio one 0 5 two 1 6 three 2 7Colorado one 3 8 two 4 9 three 5 10In [109]: df.unstack('state') In [110]: df.unstack('state').stack('side')Out[109]: Out[110]:side left right state Ohio Coloradostate Ohio Colorado Ohio Colorado number sidenumber one left 0 3one 0 3 5 8 right 5 8two 1 4 6 9 two left 1 4three 2 5 7 10 right 6 9 three left 2 5 right 7 10

将“长格式”旋转为“宽格式”

时间序列数据通常是以所谓的“长格式”（long）或“堆叠格式”（stacked）存储在数据库和CSV中的：译注4

In [116]: ldata[:10]Out[116]: date item value0 1959-03-31 00:00:00 realgdp 2710.3491 1959-03-31 00:00:00 infl 0.0002 1959-03-31 00:00:00 unemp 5.8003 1959-06-30 00:00:00 realgdp 2778.8014 1959-06-30 00:00:00 infl 2.3405 1959-06-30 00:00:00 unemp 5.1006 1959-09-30 00:00:00 realgdp 2775.4887 1959-09-30 00:00:00 infl 2.7408 1959-09-30 00:00:00 unemp 5.3009 1959-12-31 00:00:00 realgdp 2785.204

关系型数据库（如MySQL）中的数据经常都是这样存储的，因为固定架构（即列名和数据类型）有一个好处：随着表中数据的添加或删除，item列中的值的种类能够增加或减少。在上面那个例子中，date和item通常就是主键（用关系型数据库的说法），不仅提供了关系完整性，而且提供了更为简单的查询支持。当然这也是有缺点的：长格式的数据操作起来可能不那么轻松。你可能会更喜欢DataFrame，不同的item值分别形成一列，date列中的时间值则用作索引。DataFrame的pivot方法完全可以实现这个转换：

In [117]: pivoted = ldata.pivot('date', 'item', 'value')In [118]: pivoted.head()Out[118]:item infl realgdp unempdate1959-03-31 0.00 2710.349 5.81959-06-30 2.34 2778.801 5.11959-09-30 2.74 2775.488 5.31959-12-31 0.27 2785.204 5.61960-03-31 2.31 2847.699 5.2

前两个参数值分别用作行和列索引的列名，最后一个参数值则是用于填充DataFrame的数据列的列名。假设有两个需要参与重塑的数据列：

In [119]: ldata['value2'] = np.random.randn(len(ldata))In [120]: ldata[:10]Out[120]: date item value value20 1959-03-31 00:00:00 realgdp 2710.349 1.6690251 1959-03-31 00:00:00 infl 0.000 -0.4385702 1959-03-31 00:00:00 unemp 5.800 -0.5397413 1959-06-30 00:00:00 realgdp 2778.801 0.4769854 1959-06-30 00:00:00 infl 2.340 3.2489445 1959-06-30

00:00:00 unemp 5.100 -1.0212286 1959-09-30 00:00:00 realgdp 2775.488 -0.5770877 1959-09-30 00:00:00 infl 2.740 0.1241218 1959-09-30 00:00:00 unemp 5.300 0.3026149 1959-12-31 00:00:00 realgdp 2785.204 0.523772

如果忽略最后一个参数，得到的DataFrame就会带有层次化的列：

In [121]: pivoted = ldata.pivot('date', 'item')In [122]: pivoted[:5]Out[122]: value value2item infl realgdp unemp infl realgdp unempdate1959-03-31 0.00 2710.349 5.8 -0.438570 1.669025 -0.5397411959-06-30 2.34 2778.801 5.1 3.248944 0.476985 -1.0212281959-09-30 2.74 2775.488 5.3 0.124121 -0.577087 0.3026141959-12-31 0.27 2785.204 5.6 0.000940 0.523772 1.3438101960-03-31 2.31 2847.699 5.2 -0.831154 -0.713544 -2.370232In [123]: pivoted['value'][:5]Out[123]:item infl realgdp unempdate1959-03-31 0.00 2710.349 5.81959-06-30 2.34 2778.801 5.11959-09-30 2.74 2775.488 5.31959-12-31 0.27 2785.204 5.61960-03-31 2.31 2847.699 5.2

注意，pivot其实只是一个快捷方式而已：用set_index创建层次化索引，再用unstack重塑。

In [124]: unstacked = ldata.set_index(['date', 'item']).unstack('item')In [125]: unstacked[:7]Out[125]: value value2item infl realgdp unemp infl realgdp unempdate1959-03-31 0.00 2710.349 5.8 -0.438570 1.669025 -0.5397411959-06-30 2.34 2778.801 5.1 3.248944 0.476985 -1.0212281959-09-30 2.74 2775.488 5.3 0.124121 -0.577087 0.3026141959-12-31 0.27 2785.204 5.6 0.000940 0.523772 1.3438101960-03-31 2.31 2847.699 5.2 -0.831154 -0.713544 -2.3702321960-06-30 0.14 2834.390 5.2 -0.860757 -1.860761 0.5601451960-09-30 2.70 2839.022 5.6 0.119827 -1.265934 -1.063512

译注4

：由于作者在此处并未介绍ldata的生成代码，而后面又需要用到，所以不能独立看待这段代码。下载的资料采用的不是这个格式，需要处理一下才可用。如果不会处理或觉得太麻烦，就用Excel编辑一下吧。不过还是建议处理一下，就当做练手了。给个相对比较简单的小提示：先加载进来，然后stack，然后保存，然后再加载进来。

数据转换

本章到目前为止介绍的都是数据的重排。另一类重要操作则是过滤、清理以及其他的转换工作。

移除重复数据

DataFrame中常常会出现重复行。下面就是一个例子：

In [126]: data = DataFrame({'k1': ['one'] *3 + ['two']* 4, ...: 'k2': [1, 1, 2, 3, 3, 4, 4]})In [127]: dataOut[127]: k1 k20 one 11 one 12 one 23 two 34 two 35 two 46 two 4

DataFrame的duplicated方法返回一个布尔型Series，表示各行是否是重复行：

In [128]: data.duplicated()Out[128]:0 False1 True2 False3 False4 True5 False6 True

还有一个与此相关的drop*duplicates*方法，它用于返回一个移除了重复行的*Data-Frame[*译注*5](#809468440711498-YiZhu5Yuan_Wen_Zhe_Li_De_Yi_Si_Hen_You_Wen_Ti**Yuan_Wen_Shuo_De_Shi**Fan_Hui_duplicatedWei_TrueDe_DataFrame*)：

In [129]: data.drop_duplicates()Out[129]: k1 k20 one 12 one 23 two 35 two 4

这两个方法默认会判断全部列，你也可以指定部分列进行重复项判断。假设你还有一列值，且只希望根据k1列过滤重复项：

In [130]: data['v1'] = range(7)In [131]: data.drop_duplicates(['k1'])Out[131]: k1 k2 v10 one 1 03 two 3 3

duplicated和drop_duplicates默认保留的是第一个出现的值组合。传入take_last=True则保留最后一个：

In [132]: data.drop_duplicates(['k1', 'k2'], take_last=True)Out[132]: k1 k2 v11 one 1 12 one 2 24 two 3 46 two 4 6

利用函数或映射进行数据转换

在对数据集进行转换时，你可能希望根据数组、Series或DataFrame列中的值来实现该转换工作。我们来看看下面这组有关肉类的数据：

In [133]: data = DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastrami', ...: 'corned beef', 'Bacon', 'pastrami', 'honey ham', ...: 'nova lox'], ...: 'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})In [134]: dataOut[134]: food ounces0 bacon 4.01 pulled pork 3.02 bacon 12.03 Pastrami 6.04 corned beef 7.55 Bacon 8.06 pastrami 3.07 honey ham 5.08 nova lox 6.0

假设你想要添加一列表示该肉类食物来源的动物类型。我们先编写一个肉类到动物的映射：

meat_to_animal = {   'bacon': 'pig',   'pulled pork': 'pig',   'pastrami': 'cow',   'corned beef': 'cow',   'honey ham': 'pig',   'nova lox': 'salmon'}

Series的map方法可以接受一个函数或含有映射关系的字典型对象，但是这里有一个小问题，即有些肉类的首字母大写了，而另一些则没有。因此，我们还需要将各个值转换为小写：

    In [136]: data['animal'] = data['food'].map(str.lower).map(meat_to_animal)   In [137]: data   Out[137]:   food ounces animal   0 bacon 4.0 pig   1 pulled pork 3.0 pig   2 bacon 12.0 pig   3 Pastrami 6.0 cow   4 corned beef 7.5 cow   5 Bacon 8.0 pig   6 pastrami 3.0 cow   7 honey ham 5.0 pig   8 nova lox 6.0 salmon

我们也可以传入一个能够完成全部这些工作的函数：

In [138]: data['food'].map(lambda x: meat_to_animal[x.lower()]) Out[138]: 0 pig 1 pig 2 pig 3 cow 4 cow 5 pig 6 cow 7 pig 8 salmon Name: food

使用map是一种实现元素级转换以及其他数据清理工作的便捷方式。

替换值

利用fillna方法填充缺失数据可以看做值替换的一种特殊情况。虽然前面提到的map可用于修改对象的数据子集，而replace则提供了一种实现该功能的更简单、更灵活的方式。我们来看看下面这个Series：

In [139]: data = Series([1., -999., 2., -999., -1000., 3.]) In [140]: data Out[140]: 0 1 1 -999 2 2 3 -999 4 -1000 5 3

-999这个值可能是一个表示缺失数据的标记值。要将其替换为pandas能够理解的NA值，我们可以利用replace来产生一个新的Series：

In [141]: data.replace(-999, np.nan) Out[141]: 0 1 1 NaN 2 2 3 NaN 4 -1000 5 3

如果你希望一次性替换多个值，可以传入一个由待替换值组成的列表以及一个替换值：

In [142]: data.replace([-999, -1000], np.nan) Out[142]: 0 1 1 NaN 2 2 3 NaN 4 NaN 5 3

如果希望对不同的值进行不同的替换，则传入一个由替换关系组成的列表即可：

In [143]: data.replace([-999, -1000], [np.nan, 0]) Out[143]: 0 1 1 NaN 2 2 3 NaN 4 0 5 3

传入的参数也可以是字典：

In [144]: data.replace({-999: np.nan, -1000: 0}) Out[144]: 0 1 1 NaN 2 2 3 NaN 4 0 5 3

重命名轴索引

跟Series中的值一样，轴标签也可以通过函数或映射进行转换，从而得到一个新对象。轴还可以被就地修改，而无需新建一个数据结构。接下来看看下面这个简单的例子：

In [145]: data = DataFrame(np.arange(12).reshape((3, 4)), ...: index=['Ohio', 'Colorado', 'New York'], ...: columns=['one', 'two', 'three', 'four'])

跟Series一样，轴标签也有一个map方法：

In [146]: data.index.map(str.upper) Out[146]: array([OHIO, COLORADO, NEW YORK], dtype=object)

你可以将其赋值给index，这样就可以对DataFrame进行就地修改了：

In [147]: data.index = data.index.map(str.upper) In [148]: data Out[148]:   one two three four OHIO 0 1 2 3 COLORADO 4 5 6 7 NEW YORK 8 9 10 11

如果想要创建数据集的转换版（而不是修改原始数据），比较实用的方法是rename：

In [149]: data.rename(index=str.title, columns=str.upper) Out[149]:   ONE TWO THREE FOUR Ohio 0 1 2 3 Colorado 4 5 6 7 New York 8 9 10 11

特别说明一下，rename可以结合字典型对象实现对部分轴标签的更新：

In [150]: data.rename(index={'OHIO': 'INDIANA'},   ...: columns={'three': 'peekaboo'}) Out[150]:   one two peekaboo four INDIANA 0 1 2 3 COLORADO 4 5 6 7 NEW YORK 8 9 10 11

rename帮我们实现了：复制DataFrame并对其索引和列标签进行赋值。如果希望就地修改某个数据集，传入inplace=True即可：

总是返回DataFrame的引用 In [151]: _ = data.rename(index={'OHIO': 'INDIANA'}, inplace=True) In [152]: data Out[152]:   one two three four INDIANA 0 1 2 3 COLORADO 4 5 6 7 NEW YORK 8 9 10 11

离散化和面元划分

为了便于分析，连续数据常常被离散化或拆分为“面元”（bin）。假设有一组人员数据，而你希望将它们划分为不同的年龄组：

In [153]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]

接下来将这些数据划分为“18到25”、“26到35”、“35到60”以及“60以上”几个面元。要实现该功能，你需要使用pandas的cut函数：

In [154]: bins = [18, 25, 35, 60, 100] In [155]: cats = pd.cut(ages, bins) In [156]: cats Out[156]: Categorical: array([(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], (18, 25],   (35, 60], (25, 35], (60, 100], (35, 60], (35, 60], (25, 35]], dtype=object) Levels (4): Index([(18, 25], (25, 35], (35, 60], (60, 100]], dtype=object)

pandas返回的是一个特殊的Categorical对象。你可以将其看做一组表示面元名称的字符串。实际上，它含有一个表示不同分类名称的levels数组以及一个为年龄数据进行标号的labels属性：

In [157]: cats.labels Out[157]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1]) In [158]: cats.levels Out[158]: Index([(18, 25], (25, 35], (35, 60], (60, 100]], dtype=object) In [159]: pd.value_counts(cats) Out[159]: (18, 25] 5 (35, 60] 3 (25, 35] 3 (60, 100] 1

跟“区间”的数学符号一样，圆括号表示开端，而方括号则表示闭端（包括）。哪边是闭端可以通过right=False进行修改：

In [160]: pd.cut(ages, [18, 26, 36, 61, 100], right=False) Out[160]: Categorical: array([[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), [18, 26), [36, 61), [26, 36), [61, 100), [36, 61), [36, 61), [26, 36)], dtype=object) Levels (4): Index([[18, 26), [26, 36), [36, 61), [61, 100)], dtype=object)

你也可以设置自己的面元名称，将labels选项设置为一个列表或数组即可：

In [161]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior'] In [162]: pd.cut(ages, bins, labels=group_names) Out[162]: Categorical: array([Youth, Youth, Youth, YoungAdult, Youth, Youth, MiddleAged, YoungAdult, Senior, MiddleAged, MiddleAged, YoungAdult], dtype=object) Levels (4): Index([Youth, YoungAdult, MiddleAged, Senior], dtype=object)

如果向cut传入的是面元的数量而不是确切的面元边界，则它会根据数据的最小值和最大值计算等长面元。下面这个例子中，我们将一些均匀分布的数据分成四组：

In [163]: data = np.random.rand(20) In [164]: pd.cut(data, 4, precision=2) Out[164]: Categorical: array([(0.45, 0.67], (0.23, 0.45], (0.0037, 0.23], (0.45, 0.67], (0.67, 0.9], (0.45, 0.67], (0.67, 0.9], (0.23, 0.45], (0.23, 0.45], (0.67, 0.9], (0.67, 0.9], (0.67, 0.9], (0.23, 0.45], (0.23, 0.45], (0.23, 0.45], (0.67, 0.9], (0.0037, 0.23], (0.0037, 0.23], (0.23, 0.45], (0.23, 0.45]], dtype=object) Levels (4): Index([(0.0037, 0.23], (0.23, 0.45], (0.45, 0.67], (0.67, 0.9]], dtype=object)

qcut是一个非常类似于cut的函数，它可以根据样本分位数对数据进行面元划分。根据数据的分布情况，cut可能无法使各个面元中含有相同数量的数据点。而qcut由于使用的是样本分位数，因此可以得到大小基本相等的面元：

In [165]: data = np.random.randn(1000) # 正态分布 In [166]: cats = pd.qcut(data, 4) # 按四分位数进行切割 In [167]: cats Out[167]: Categorical: array([(-0.022, 0.641], [-3.745, -0.635], (0.641, 3.26], ..., (-0.635, -0.022], (0.641, 3.26], (-0.635, -0.022]], dtype=object) Levels (4): Index([[-3.745, -0.635], (-0.635, -0.022], (-0.022, 0.641], (0.641, 3.26]], dtype=object) In [168]: pd.value_counts(cats) Out[168]: [-3.745, -0.635] 250 (0.641, 3.26] 250 (-0.635, -0.022] 250 (-0.022, 0.641] 250

跟cut一样，也可以设置自定义的分位数（0到1之间的数值，包含端点）：

In [169]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.]) Out[169]: Categorical: array([(-0.022, 1.302], (-1.266, -0.022], (-0.022, 1.302], ..., (-1.266, -0.022], (-0.022, 1.302], (-1.266, -0.022]], dtype=object) Levels (4): Index([[-3.745, -1.266], (-1.266, -0.022], (-0.022, 1.302], (1.302, 3.26]], dtype=object)

本章稍后在讲解聚合和分组运算时会再次用到cut和qcut，因为这两个离散化函数对分量和分组分析非常重要。

检测和过滤异常值

异常值译注6（outlier）的过滤或变换运算在很大程度上其实就是数组运算。来看一个含有正态分布数据的DataFrame：

In [170]: np.random.seed(12345) In [171]: data = DataFrame(np.random.randn(1000, 4)) In [172]: data.describe() Out[172]:   0 1 2 3 count 1000.000000 1000.000000 1000.000000 1000.000000 mean -0.067684 0.067924 0.025598 -0.002298 std 0.998035 0.992106 1.006835 0.996794 min -3.428254 -3.548824 -3.184377 -3.745356 25% -0.774890 -0.591841 -0.641675 -0.644144 50% -0.116401 0.101143 0.002073 -0.013611 75% 0.616366 0.780282 0.680391 0.654328 max 3.366626 2.653656 3.260383 3.927528

假设你想要找出某列中绝对值大小超过3的值：

In [173]: col = data[3] In [174]: col[np.abs(col) > 3] Out[174]: 97 3.927528 305 -3.399312 400 -3.745356 Name: 3

要选出全部含有“超过3或－3的值”的行，你可以利用布尔型DataFrame以及any方法：

In [175]: data[(np.abs(data) > 3).any(1)] Out[175]:   0 1 2 3 5 -0.539741 0.476985 3.248944 -1.021228 97 -0.774363 0.552936 0.106061 3.927528 102 -0.655054 -0.565230 3.176873 0.959533 305 -2.315555 0.457246 -0.025907 -3.399312 324 0.050188 1.951312 3.260383 0.963301 400 0.146326 0.508391 -0.196713 -3.745356 499 -0.293333 -0.242459 -3.056990 1.918403 523 -3.428254 -0.296336 -0.439938 -0.867165 586 0.275144 1.179227 -3.184377 1.369891 808 -0.362528 -3.548824 1.553205 -2.186301 900 3.366626 -2.372214 0.851010 1.332846

根据这些条件，即可轻松地对值进行设置。下面的代码可以将值限制在区间－3到3以内：

In [176]: data[np.abs(data) > 3] = np.sign(data) * 3 In [177]: data.describe() Out[177]: 0 1 2 3 count 1000.000000 1000.000000 1000.000000 1000.000000 mean -0.067623 0.068473 0.025153 -0.002081 std 0.995485 0.990253 1.003977 0.989736 min -3.000000 -3.000000 -3.000000 -3.000000 25% -0.774890 -0.591841 -0.641675 -0.644144 50% -0.116401 0.101143 0.002073 -0.013611 75% 0.616366 0.780282 0.680391 0.654328 max 3.000000 2.653656 3.000000 3.000000

np.sign这个ufunc返回的是一个由1和－1组成的数组，表示原始值的符号。

排列和随机采样

利用numpy.random.permutation函数可以轻松实现对Series或DataFrame的列的排列工作（permuting，随机重排序译注7）。通过需要排列的轴的长度调用permutation，可产生一个表示新顺序的整数数组：

In [178]: df = DataFrame(np.arange(5 * 4).reshape(5, 4))In [179]: sampler = np.random.permutation(5)In [180]: samplerOut[180]: array([1, 0, 2, 3, 4])

然后就可以在基于ix的索引操作或take函数中使用该数组了：

In [181]: dfOut[181]: 0 1 2 30 0 1 2 31 4 5 6 72 8 9 10 113 12 13 14 154 16 17 18 19In [182]: df.take(sampler)Out[182]: 0 1 2 31 4 5 6 70 0 1 2 32 8 9 10 113 12 13 14 154 16 17 18 19

如果不想用替换的方式选取随机子集，则可以使用permutation：从permutation返回的数组中切下前k个元素，其中k为期望的子集大小。虽然有很多高效的算法可以实现非替换式采样，但是手边就有的工具为什么不用呢？

In [183]: df.take(np.random.permutation(len(df))[:3])Out[183]: 0 1 2 31 4 5 6 73 12 13 14 154 16 17 18 19

要通过替换的方式产生样本，最快的方式是通过np.random.randint得到一组随机整数：

In [184]: bag = np.array([5, 7, -1, 6, 4])In [185]: sampler = np.random.randint(0, len(bag), size=10)In [186]: samplerOut[186]: array([4, 4, 2, 2, 2, 0, 3, 0, 4, 1])In [187]: draws = bag.take(sampler)In [188]: drawsOut[188]: array([4, 4, -1, -1, -1, 5, 6, 5, 4, 7])

计算指标/哑变量

另一种常用于统计建模或机器学习的转换方式是：将分类变量（categorical variable）转换为“哑变量矩阵”（dummy matrix）或“指标矩阵”（indicator matrix）。如果DataFrame的某一列中含有k个不同的值，则可以派生出一个k列矩阵或DataFrame（其值全为1和0）。pandas有一个get_dummies函数可以实现该功能（其实自己动手做一个也不难）。拿之前的一个例子来说：

In [189]: df = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'], ...: 'data1': range(6)})In [190]: pd.get_dummies(df['key'])Out[190]: a b c0 0 1 01 0 1 02 1 0 03 0 0 14 1 0 05 0 1 0

有时候，你可能想给指标DataFrame的列加上一个前缀，以便能够跟其他数据进行合并。get_dummies的prefix参数可以实现该功能：

In [191]: dummies = pd.get_dummies(df['key'], prefix='key')In [192]: df_with_dummy = df[['data1']].join(dummies)In [193]: df_with_dummyOut[193]: data1 key_a key_b key_c0 0 0 1 01 1 0 1 02 2 1 0 03 3 0 0 14 4 1 0 05 5 0 1 0

如果DataFrame中的某行同属于多个分类，则事情就会有点复杂。回到本书前面那个MovieLens 1M数据集上：译注8

In [194]: mnames = ['movie_id', 'title', 'genres']In [195]: movies = pd.read_table('ch02/movielens/movies.dat', sep='::', header=None, ...: names=mnames)In [196]: movies[:10]Out[196]: movie_id title genres0 1 Toy Story (1995) Animation|Children's|Comedy1 2 Jumanji (1995) Adventure|Children's|Fantasy2 3 Grumpier Old Men (1995) Comedy|Romance3 4 Waiting to Exhale (1995) Comedy|Drama4 5 Father

of the Bride Part II (1995) Comedy5 6 Heat (1995) Action|Crime|Thriller6 7 Sabrina (1995) Comedy|Romance7 8 Tom and Huck (1995) Adventure|Children's8 9 Sudden Death (1995) Action9 10 GoldenEye (1995) Action|Adventure|Thriller

要为每个genre添加指标变量就需要做一些数据规整操作。首先，我们从数据集中抽取出不同的genre值（注意巧用set.union）：

In [197]: genre_iter = (set(x.split('|')) for x in movies.genres)In [198]: genres = sorted(set.union(*genre_iter))

现在，我们从一个全零DataFrame开始构建指标DataFrame：

In [199]: dummies = DataFrame(np.zeros((len(movies), len(genres))), columns=genres)

接下来，迭代每一部电影并将dummies各行的项设置为1：

In [200]: for i, gen in enumerate(movies.genres): ...: dummies.ix[i, gen.split('|')] = 1

然后，再将其与movies合并起来：

In [201]: movies*windic = movies.join(dummies.add_prefix('Genre*'))In [202]: movies_windic.ix[0]Out[202]:movie_id 1title Toy Story (1995)genres Animation|Children's|ComedyGenre_Action 0Genre_Adventure 0Genre_Animation 1Genre_Children's 1Genre_Comedy 1Genre_Crime 0Genre_Documentary 0Genre_Drama 0Genre_Fantasy 0Genre_Film-Noir 0Genre_Horror 0Genre_Musical 0Genre_Mystery 0Genre_Romance 0Genre_Sci-Fi 0Genre_Thriller 0Genre_War 0Genre_Western 0Name: 0

注意： 对于很大的数据，用这种方式构建多成员指标变量就会变得非常慢。肯定需要编写一个能够利用DataFrame内部机制的更低级的函数才行。

一个对统计应用有用的秘诀是：结合get_dummies和诸如cut之类的离散化函数。

In [204]: values = np.random.rand(10)In [205]: valuesOut[205]:array([0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645, 0.6532, 0.7489, 0.6536])In [206]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]In [207]: pd.get_dummies(pd.cut(values, bins))Out[207]: (0, 0.2] (0.2, 0.4] (0.4, 0.6] (0.6, 0.8] (0.8, 1]0 0 0 0 0 11 0 1 0 0 02 1 0 0 0 03 0 1 0 0 04 0 0 1 0 05 0 0 1 0 06 0 0 0 0 17 0 0 0 1 08 0 0 0 1 09 0 0 0 1 0

译注5

：原文这里的意思很有问题，原文说的是“返回duplicated为True的DataFrame”，实际上应该是删除了duplicated为True的那些行，因此最终得到的DataFrame的duplicated不可能再含有True了。

译注6

：也叫孤立点或离群值。

译注7

：也就是中学学的那个排列，只不过不是算出所有排列，而是其中之一。

译注8

：这个数据集不在ch07中，而在ch02里面。

字符串操作

Python能够成为流行的数据处理语言，部分原因是其简单易用的字符串和文本处理功能。大部分文本运算都直接做成了字符串对象的内置方法。对于更为复杂的模式匹配和文本操作，则可能需要用到正则表达式。pandas对此进行了加强，它使你能够对整组数据应用字符串表达式和正则表达式，而且能处理烦人的缺失数据。

字符串对象方法

对于大部分字符串处理应用而言，内置的字符串方法已经能够满足要求了。例如，以逗号分隔的字符串可以用split拆分成数段：

In [208]: val = 'a,b, guido'In [209]: val.split(',')Out[209]: ['a', 'b', ' guido']

split常常结合strip（用于修剪空白符（包括换行符））一起使用：

In [210]: pieces = [x.strip() for x in val.split(',')]In [211]: piecesOut[211]: ['a', 'b', 'guido']

利用加法，可以将这些子字符串以双冒号分隔符的形式连接起来：译注9

In [212]: first, second, third = piecesIn [213]: first + '::' + second + '::' + thirdOut[213]: 'a::b::guido'

但这种方式并不是很实用。一种更快更符合Python风格的方式是，向字符串"::"的join方法传入一个列表或元组：

In [214]: '::'.join(pieces)Out[214]: 'a::b::guido'

另一类方法关注的是子串定位。检测子串的最佳方式是利用Python的in关键字（当然还可以使用index和find）：

In [215]: 'guido' in valOut[215]: TrueIn [216]: val.index(',')Out[216]: 1In [217]: val.find(':')Out[217]: -1

注意find和index的区别：如果找不到字符串，index将会引发一个异常（而不是返回－1）：

In [218]: val.index(':')---ValueError Traceback (most recent call last) in <module>()----> 1 val.index(':')ValueError: substring not found

此外还有一个count函数，它可以返回指定子串的出现次数：

In [219]: val.count(',')Out[219]: 2

replace用于将指定模式替换为另一个模式。它也常常用于删除模式：传入空字符串。

In [220]: val.replace(',', '::')Out[220]: 'a::b:: guido'In [221]: val.replace(',', '')Out[221]: 'ab guido'

这些运算大部分都能使用正则表达式实现（马上就会看到）。

Python内置的字符串方法如表7-3所示。

表7-3：Python内置的字符串方法

方法	说明
count	返回子串在字符串中的出现次数（非重叠）
endswith、startswith	如果字符串以某个后缀结尾（以某个前缀开头），则返回True
join	将字符串用作连接其他字符串序列的分隔符
index	如果在字符串中找到子串，则返回子串第一个字符所在的位置。如果没有找到，则引发ValueError。
find	如果在字符串中找到子串，则返回第一个发现的子串的第一个字符所在的位置。如果没有找到，则返回 −1
rfind	如果在字符串中找到子串，则返回最后一个发现的子串的第一个字符所在的位置。如果没有找到，则返回 −1
replace	用另一个字符串替换指定子串

表7-3：Python内置的字符串方法（续）

方法	说明
strip、rstrip、lstrip	去除空白符（包括换行符）。相当于对各个元素执行x.strip()（以及rstrip、lstrip）。译注10
split	通过指定的分隔符将字符串拆分为一组子串
lower、upper	分别将字母字符转换为小写或大写
ljust、rjust	用空格（或其他字符）填充字符串的空白侧以返回符合最低宽度的字符串

译注10：这里的说法有误。字符串的各个元素不就是字符吗？这里不是矢量化的，当涉及pandas中的这几个函数的矢量版时才应该加上后面这句。

正则表达式

正则表达式（通常称作regex）提供了一种灵活的在文本中搜索或匹配字符串模式的方式。正则表达式是根据正则表达式语言编写的字符串。Python内置的re模块负责对字符串应用正则表达式。我将通过一些例子说明其使用方法。

注意： 正则表达式的编写技巧可以自成一章译注11，因此超出了本书的范围。网上可以找到许多非常不错的教程和参考资料，比如Zed Shaw的《Learn Regex The Hard Way》（http://regex.learncodethehardway.org/book/）。

re模块的函数可以分为三个大类：模式匹配、替换以及拆分。当然，它们之间是相辅相成的。一个regex描述了需要在文本中定位的一个模式，它可以用于许多目的。我们先来看一个简单的例子：假设我想要拆分一个字符串，分隔符为数量不定的一组空白符（制表符、空格、换行符等）。描述一个或多个空白符的regex是\s+：

```
In [222]: import reIn [223]: text = "foo bar\t baz \tqux"In [224]: re.split('\s+', text)Out[224]: ['foo', 'bar', 'baz', 'qux']
```

调用re.split('\s+',text)时，正则表达式会先被编译，然后再在text上调用其split方法。你可以用re.compile自己编译regex以得到一个可重用的regex对象：

```
In [225]: regex = re.compile('\s+')In [226]: regex.split(text)Out[226]: ['foo', 'bar', 'baz', 'qux']
```

如果只希望得到匹配regex的所有模式，则可以使用findall方法：

```
In [227]: regex.findall(text)Out[227]: [' ', '\t ', ' \t']
```

注意： 如果想避免正则表达式中不需要的转义（\），则可以使用原始字符串字面量如r'C:\x'（也可以编写其等价式'C:\x'）。

如果打算对许多字符串应用同一条正则表达式，强烈建议通过re.compile创建regex对象。这样将可以节省大量的CPU时间。

match和search跟findall功能类似。findall返回的是字符串中所有的匹配项，而search则只返回第一个匹配项。match更加严格，它只匹配字符串的首部。来看一个小例子，假设我们有一段文本以及一条能够识别大部分电子邮件地址的正则表达式：

```
text = """Dave dave@google.comSteve steve@gmail.comRob rob@gmail.comRyan ryan@yahoo.com"""pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+.[A-Z]{2,4}'# re.IGNORECASE的作用是使正则表达式对大小写不敏感regex = re.compile(pattern, flags=re.IGNORECASE)
```

对text使用findall将得到一组电子邮件地址：

```
In [229]: regex.findall(text)Out[229]: ['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

search返回的是文本中第一个电子邮件地址（以特殊的匹配项对象形式返回）。对于上面那个regex，匹配项对象只能告诉我们模式在原字符串中的起始和结束位置：

```
In [230]: m = regex.search(text)In [231]: mOut[231]: <_sre.SRE_Match at="" 0x10a05de00="">In [232]: text[m.start():m.end()]Out[232]: 'dave@google.com'
```

regex.match则将返回None，因为它只匹配出现在字符串开头的模式：

```
In [233]: print regex.match(text)None
```

另外还有一个sub方法，它会将匹配到的模式替换为指定字符串，并返回所得到的新字符串：

In [234]: print regex.sub('REDACTED', text)Dave REDACTEDSteve REDACTEDRob REDACTEDRyan REDACTED

假设你不仅想要找出电子邮件地址，还想将各个地址分成3个部分：用户名、域名以及域后缀。要实现此功能，只需将待分段的模式的各部分用圆括号包起来即可：

In [235]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+).([A-Z]{2,4})'In [236]: regex = re.compile(pattern, flags=re.IGNORECASE)

由这种正则表达式所产生的匹配项对象，可以通过其groups方法返回一个由模式各段组成的元组：

In [237]: m = regex.match('wesm@bright.net')In [238]: m.groups()Out[238]: ('wesm', 'bright', 'net')

对于带有分组功能的模式，findall会返回一个元组列表：

In [239]: regex.findall(text)Out[239]:[('dave', 'google', 'com'), ('steve', 'gmail', 'com'), ('rob', 'gmail', 'com'), ('ryan', 'yahoo', 'com')]

sub还能通过诸如\1、\2之类的特殊符号访问各匹配项中的分组：

In [240]: print regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text)Dave Username: dave, Domain: google, Suffix: comSteve Username: steve, Domain: gmail, Suffix: comRob Username: rob, Domain: gmail, Suffix: comRyan Username: ryan, Domain: yahoo, Suffix: com

Python中还有许多的正则表达式，但大部分都超出了本书的范围。为了给你一点感觉，我对上面那个电子邮件正则表达式做一点小变动：为各个匹配分组加上一个名称。

regex = re.compile(r""" (?P<username>[A-Z0-9._%+-]+) @ (?P<domain>[A-Z0-9.-]+) . (?P<suffix>[A-Z]{2,4})""", flags=re.IGNORECASE|re.VERBOSE)

由这种正则表达式所产生的匹配项对象可以得到一个简单易用的带有分组名称的字典：

In [242]: m = regex.match('wesm@bright.net')In [243]: m.groupdict()Out[243]: {'domain': 'bright', 'suffix': 'net', 'username': 'wesm'}

前面提及的正则表达式的方法与说明如表7-4所示。

表7-4：正则表达式方法

方法	说明
findall、finditer	返回字符串中所有的非重叠匹配模式。findall返回的是由所有模式组成的列表，而finditer则通过一个迭代器逐个返回
match	从字符串起始位置匹配模式，还可以对模式各部分进行分组。如果匹配到模式，则返回一个匹配项对象，否则返回None
search	扫描整个字符串以匹配模式。如果找到则返回一个匹配项对象。跟match不同，其匹配项可以位于字符串的任意位置，而不仅仅是起始处
split	根据找到的模式将字符串拆分为数段
sub、subn	将字符串中所有的（sub）或前n个（subn）模式替换为指定表达式[译注12]。在替换字符串中可以通过\1、\2等符号表示各分组项

译注12：这个表达式要么是字符串要么是函数返回值。

pandas中矢量化的字符串函数

清理待分析的散乱数据时，常常需要做一些字符串规整化工作。更为复杂的情况是，含有字符串的列有时还含有缺失数据：

In [244]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com', ...: 'Rob': 'rob@gmail.com', 'Wes': np.nan}In [245]: data = Series(data)In [246]: dataOut[246]:Dave dave@google.comRob rob@gmail.comSteve steve@gmail.comWes NaNIn [247]: data.isnull()Out[247]:Dave FalseRob FalseSteve FalseWes True

通过data.map，所有字符串和正则表达式方法都能被应用于（传入lambda表达式或其他函数）各个值，但是如果存在NA就会报错。为了解决这个问题，Series有一些能够跳过NA值的字符串操作方法。通过Series的str属性即可访问这些方法。例如，我们可以通过str.contains检查各个电子邮件地址是否含有"gmail"：

In [248]: data.str.contains('gmail')Out[248]:Dave FalseRob TrueSteve TrueWes NaN

这里也可以使用正则表达式，还可以加上任意re选项（如IGNORECASE）：

In [249]: patternOut[249]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'In [250]: data.str.findall(pattern, flags=re.IGNORECASE)Out[250]:Dave [('dave', 'google', 'com')]Rob [('rob', 'gmail', 'com')]Steve [('steve', 'gmail', 'com')]Wes NaN

有两个办法可以实现矢量化的元素获取操作：要么使用str.get，要么在str属性上使用索引。

In [251]: matches = data.str.match(pattern, flags=re.IGNORECASE)In [252]: matchesOut[252]:Dave ('dave', 'google', 'com')Rob ('rob', 'gmail', 'com')Steve ('steve', 'gmail', 'com')Wes NaNIn [253]: matches.str.get(1)Out[253]:Dave googleRob gmailSteve gmailWes NaNIn [254]: matches.str[0]Out[254]:Dave daveRob robSteve steveWes NaN

你可以利用下面这种代码对字符串进行子串截取：

In [255]: data.str[:5]Out[255]:Dave dave@Rob rob@gSteve steveWes NaN

表7-5介绍了矢量化的字符串方法。

表7-5：矢量化的字符串方法

方法	说明
cat	实现元素级的字符串连接操作，可指定分隔符
contains	返回表示各字符串是否含有指定模式的布尔型数组
count	模式的出现次数
endswith、startswith	相当于对各个元素执行x.endswith(pattern)或x.startswith(pattern)
findall	计算各字符串的模式列表
get	获取各元素的第i个字符
join	根据指定的分隔符将Series中各元素的字符串连接起来
len	计算各字符串的长度
lower、upper	转换大小写。相当于对各个元素执行x.lower()或x.upper()
match	根据指定的正则表达式对各个元素执行re.match
pad	在字符串的左边、右边或左右两边添加空白符
center	相当于pad(side='both')
repeat	重复值。例如，s.str.repeat(3)相当于对各个字符串执行x * 3
replace	用指定字符串替换找到的模式
slice	对Series中的各个字符串进行子串截取
split	根据分隔符或正则表达式对字符串进行拆分
strip、rstrip、lstrip	去除空白符，包括换行符。相当于对各个元素执行x.strip()、x.rstrip()、x.lstrip()

译注9

：其实什么分隔符都行，原文有歧义。

译注11

：别说一章，目前市面上专门介绍正则表达式的书非常多。

示例：USDA食品数据库

美国农业部（USDA）制作了一份有关食物营养信息的数据库。Ashley Williams（一名来自英国的技术牛人）制作了该数据的JSON版（http://ashleyw.co.uk/project/food-nutrient-database）。其中的记录如下所示：

{ "id": 21441, "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY, Wing, meat and skin with breading", "tags": ["KFC"], "manufacturer": "Kentucky Fried Chicken", "group": "Fast Foods", "portions": [{ "amount": 1, "unit": "wing, with skin", "grams": 68.0 }, ...], "nutrients": [{ "value": 20.8, "units": "g", "description": "Protein", "group": "Composition" }, ...]}

每种食物都带有若干标识性属性以及两个有关营养成分和分量的列表。这种形式的数据不是很适合分析工作，因此我们需要做一些规整化以使其具有更好用的形式。

从上面列举的那个网址下载并解压数据之后，你可以用任何喜欢的JSON库将其加载到Python中。我用的是Python内置的json模块：

In [256]: import jsonIn [257]: db = json.load(open('ch07/foods-2011-10-03.json'))In [258]: len(db)Out[258]: 6636

db中的每个条目都是一个含有某种食物全部数据的字典。nutrients字段是一个字典列表，其中的每个字典对应一种营养成分：

In [259]: db[0].keys()Out[259]:[u'portions', u'description', u'tags', u'nutrients', u'group', u'id', u'manufacturer']In [260]: db[0]['nutrients'][0]Out[260]:{u'description': u'Protein', u'group': u'Composition', u'units': u'g', u'value': 25.18}In [261]: nutrients = DataFrame(db[0]['nutrients'])In [262]: nutrients[:7]Out[262]: description group units value0 Protein Composition g 25.181 Total lipid (fat) Composition g 29.202 Carbohydrate, by difference Composition g 3.063 Ash Other g 3.284 Energy Energy kcal 376.005 Water Composition g 39.286 Energy Energy kJ 1573.00

在将字典列表转换为DataFrame时，可以只抽取其中的一部分字段。这里，我们将取出食物的名称、分类、编号以及制造商等信息：

In [263]: info_keys = ['description', 'group', 'id', 'manufacturer']In [264]: info = DataFrame(db, columns=info_keys)In [265]: info[:5]Out[265]: description group id manufacturer0 Cheese, caraway Dairy and Egg Products 10081 Cheese, cheddar Dairy and Egg Products 10092 Cheese, edam Dairy and Egg Products 10183 Cheese, feta Dairy and Egg Products 10194 Cheese, mozzarella, part skim milk Dairy and Egg Products 1028In [266]: infoOut[266]:Int64Index: 6636 entries, 0 to 6635Data columns:description 6636 non-null valuesgroup 6636 non-null valuesid 6636 non-null valuesmanufacturer 5195 non-null valuesdtypes: int64(1), object(3)

通过value_counts，你可以查看食物类别的分布情况：

In [267]: pd.value_counts(info.group)[:10]Out[267]:Vegetables and Vegetable Products 812Beef Products 618Baked Products 496Breakfast Cereals 403Legumes and Legume Products 365Fast Foods 365Lamb, Veal, and Game Products 345Sweets 341Pork Products 328Fruits and Fruit Juices 328

现在，为了对全部营养数据做一些分析，最简单的办法是将所有食物的营养成分整合到一个大表中。我们分几个步骤来实现该目的。首先，将各食物的营养成分列表转换为一个DataFrame，并添加一个表示编号的列，然后将该DataFrame添加到一个列表中。最后通过concat将这些东西连接起来就可以了：

nutrients = []for rec in db: fnuts = DataFrame(rec['nutrients']) fnuts['id'] = rec['id'] nutrients.append(fnuts)nutrients = pd.concat(nutrients, ignore_index=True)

如果一切顺利的话，nutrients应该是下面这样的：

In [269]: nutrientsOut[269]:Int64Index: 389355 entries, 0 to 389354Data columns:description 389355 non-null valuesgroup 389355 non-null valuesunits 389355 non-null valuesvalue 389355 non-null valuesid 389355 non-null valuesdtypes: float64(1), int64(1), object(3)

我发现这个DataFrame中无论如何都会有一些重复项，所以直接丢弃就可以了：

In [270]: nutrients.duplicated().sum()Out[270]: 14179In [271]: nutrients = nutrients.drop_duplicates()

由于两个DataFrame对象中都有"group"和"description"，所以为了明确到底谁是谁，我们需要对它们进行重命名：

In [272]: col_mapping = {'description' : 'food', ...: 'group' : 'fgroup'}In [273]: info = info.rename(columns=col_mapping, copy=False)In [274]: infoOut[274]:Int64Index: 6636 entries, 0 to 6635Data columns:food 6636 non-null valuesfgroup 6636 non-null valuesid 6636 non-null valuesmanufacturer 5195 non-null valuesdtypes: int64(1), object(3)In [275]: col_mapping = {'description' : 'nutrient', ...: 'group' : 'nutgroup'}In [276]: nutrients = nutrients.rename(columns=col_mapping, copy=False)In [277]: nutrientsOut[277]:Int64Index: 375176 entries, 0 to 389354Data columns:nutrient 375176 non-null valuesnutgroup 375176 non-null valuesunits 375176 non-null valuesvalue 375176 non-null valuesid 375176 non-null valuesdtypes: float64(1), int64(1), object(3)

做完这些事情之后，就可以将info跟nutrients合并起来：

In [278]: ndata = pd.merge(nutrients, info, on='id', how='outer')In [279]: ndataOut[279]:Int64Index: 375176 entries, 0 to 375175Data columns:nutrient 375176 non-null valuesnutgroup 375176 non-null valuesunits 375176 non-null valuesvalue 375176 non-null valuesid 375176 non-null valuesfood 375176 non-null valuesfgroup 375176 non-null valuesmanufacturer 293054 non-null valuesdtypes: float64(1), int64(1), object(6)In [280]: ndata.ix[30000]Out[280]:nutrient Folic acidnutgroup Vitaminsunits mcgvalue 0id 5658food Ostrich, top loin, cookedfgroup Poultry ProductsmanufacturerName: 30000

接下来的两章中将介绍切片和切块、聚合、图形化方面的工具，所以在你掌握了那些方法之后可以再用这个数据集来练练手。比如说，我们可以根据食物分类和营养类型画出一张中位值图（如图7-1所示）：

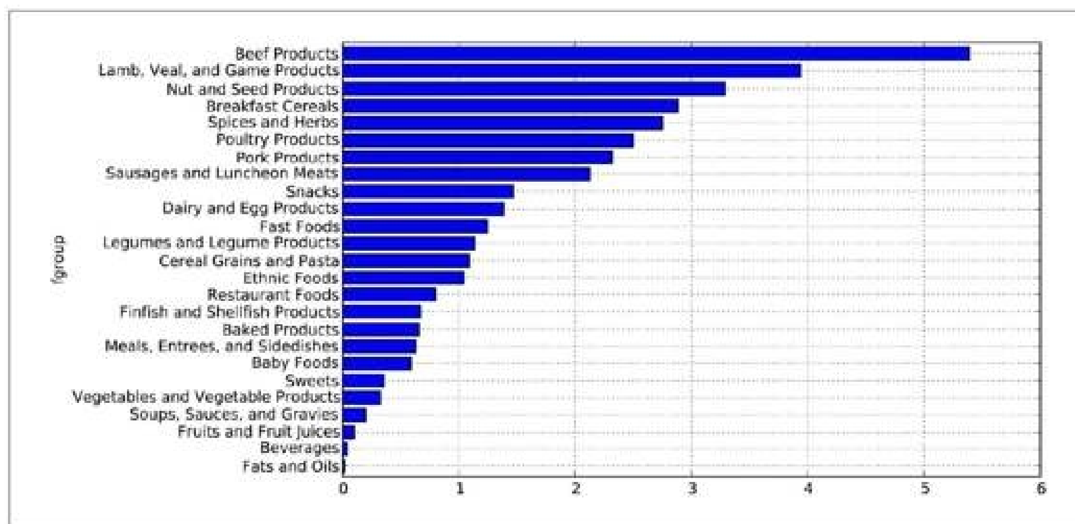


图7-1：根据营养分类得出的锌中位值

In [281]: result = ndata.groupby(['nutrient', 'fgroup'])['value'].quantile(0.5)In [282]: result['Zinc, Zn'].order().plot(kind='barh')

只要稍微动一动脑子，就可以发现各营养成分最为丰富的食物是什么了：

by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])get_maximum = lambda x: x.xs(x.value.idxmax())get_minimum = lambda x: x.xs(x.value.idxmin())max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]# 让food小一点max_foods.food = max_foods.food.str[:50]

由于得到的DataFrame很大，所以不方便在书里面全部打印出来。这里只给出"Amino Acids"营养分组：

In [284]: max_foods.ix['Amino Acids']['food']Out[284]:nutrientAlanine Gelatins, dry powder, unsweetenedArginine Seeds, sesame flour, low-fatAspartic acid Soy protein isolateCystine Seeds, cottonseed flour, low fat (glandless)Glutamic acid Soy protein isolateGlycine Gelatins, dry powder, unsweetenedHistidine Whale, beluga, meat, dried (Alaska Native)Hydroxyproline KENTUCKY FRIED CHICKEN, Fried Chicken, ORIGINAL RIsoleucine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNALeucine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNALysine Seal, bearded (Oogruk), meat, dried (Alaska NativMethionine Fish, cod, Atlantic, dried and saltedPhenylalanine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAProline Gelatins, dry powder, unsweetenedSerine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAThreonine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNATryptophan Sea lion, Steller, meat with fat (Alaska Native)Tyrosine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAValine Soy protein isolate, PROTEIN TECHNOLOGIES INTERNAName: food

第8章　绘图和可视化

绘图是数据分析工作中最重要的任务之一，是探索过程的一部分，例如，帮助我们找出异常值、必要的数据转换、得出有关模型的idea等。此外，还可以利用诸如d3.js（http://d3js.org/）之类的工具为Web应用构建交互式图像。Python有许多可视化工具（参见本章末尾），但是我主要讲解matplotlib（http://matplotlib.sourceforge.net）。

matplotlib是一个用于创建出版质量图表的桌面绘图包（主要是2D方面）。该项目是由John Hunter于2002年启动的，其目的是为Python构建一个MATLAB式的绘图接口。从那时起，John Hunter、Fernando Pérez（IPython的创始人）等许多人就一起合作，共同致力于将IPython和matplotlib结合起来以提供一种功能丰富且高效的科学计算环境。如果结合使用一种GUI工具包（如IPython），matplotlib还具有诸如缩放和平移等交互功能。它不仅支持各种操作系统上许多不同的GUI后端，而且还能将图片导出为各种常见的矢量（vector）和光栅（raster）图：PDF、SVG、JPG、PNG、BMP、GIF等。本书中的大部分图形都是用它生成的。

matplotlib还有许多插件工具集，如用于3D图形的mplot3d以及用于地图和投影的basemap。我将在本章末尾介绍一个利用basemap在地图上绘制数据和读取shapefiles的例子。

要使用本章中的代码示例，请确保你的IPython是以Pylab模式启动的（ipython --pylab），或通过%gui魔术命令打开了GUI事件循环集成。

matplotlib API入门

使用matplotlib的办法有很多种，最常用的方式是Pylab模式的IPython（ipython --pylab）。这样会将IPython配置为使用你所指定的matplotlib GUI后端（Tk、wxPython、PyQt、Mac OS X native、GTK）。对大部分用户而言，默认的后端就已经够用了。Pylab模式还会向IPython引入一大堆模块和函数以提供一种更接近于MATLAB的界面（见图8-1）。绘制一张简单的图表即可测试是否一切准备就绪：

```
plot(np.arange(10))
```

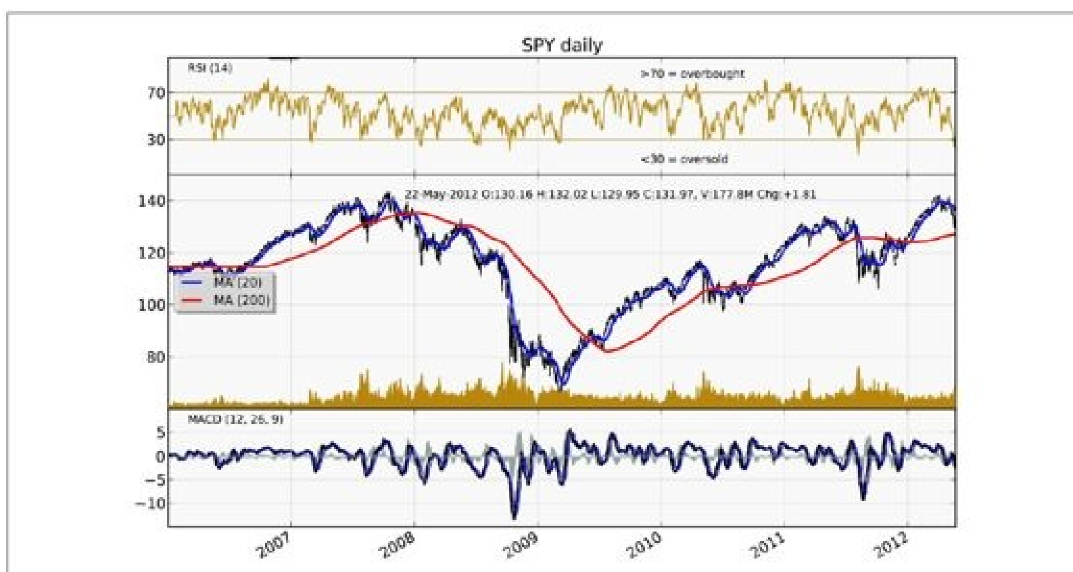


图8-1：一张比较复杂的matplotlib金融曲线图

如果一切都没有问题，就会弹出一个新窗口，其中绘制的是一条直线。你可以用鼠标或输入close()来关闭它。matplotlib API函数（如plot和close）都位于matplotlib.pyplot模块中，其通常的引入约定是：

import matplotlib.pyplot as plt

虽然pandas的绘图函数（稍后介绍）能够处理许多普通的绘图任务，但如果需要自定义一些高级功能的话就必须学习matplotlib API。

注意：虽然本书没有详细地讨论matplotlib的各种功能，但足以将你引入门。matplotlib的示例库和文档是成为绘图高手的最佳学习资源。

Figure和Subplot

matplotlib的图像都位于Figure对象中。你可以用plt.figure创建一个新的Figure：

In [13]: fig = plt.figure()

这时会弹出一个空窗口。plt.figure有一些选项，特别是figsize，它用于确保当图片保存到磁盘时具有一定的大小和纵横比。matplotlib中的Figure还支持一种MATLAB式的编号架构（例如plt.figure(2)）。通过plt.gcf()即可得到当前Figure的引用。

不能通过空Figure绘图。必须用add_subplot创建一个或多个subplot才行：

In [14]: ax1 = fig.add_subplot(2, 2, 1)

这条代码的意思是：图像应该是2×2的，且当前选中的是4个subplot中的第一个（编号从1开始）。如果再把后面两个subplot也创建出来，最终得到的图像如图8-2所示。

In [15]: ax2 = fig.add_subplot(2, 2, 2)In [16]: ax3 = fig.add_subplot(2, 2, 3)

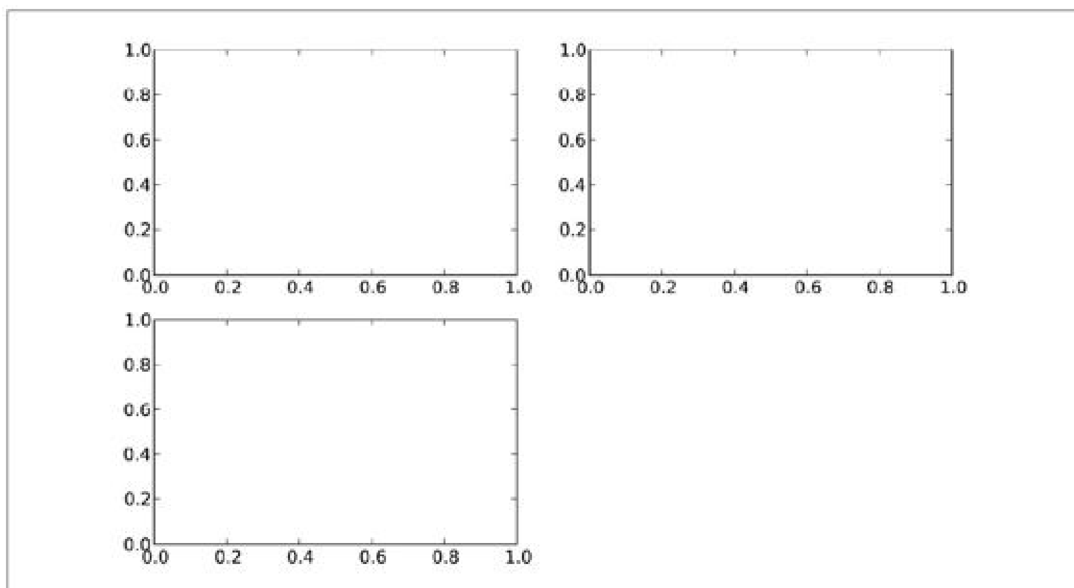


图8-2：带有三个subplot的Figure

如果这时发出一条绘图命令（如plt.plot([1.5,3.5,-2,1.6])），matplotlib就会在最后一个用过的subplot（如果没有则创建一个）上进行绘制。因此，如果我们执行下列命令，你就会得到如图8-3所示的结果：

In [17]: from numpy.random import randnIn [18]: plt.plot(randn(50).cumsum(), 'k--')

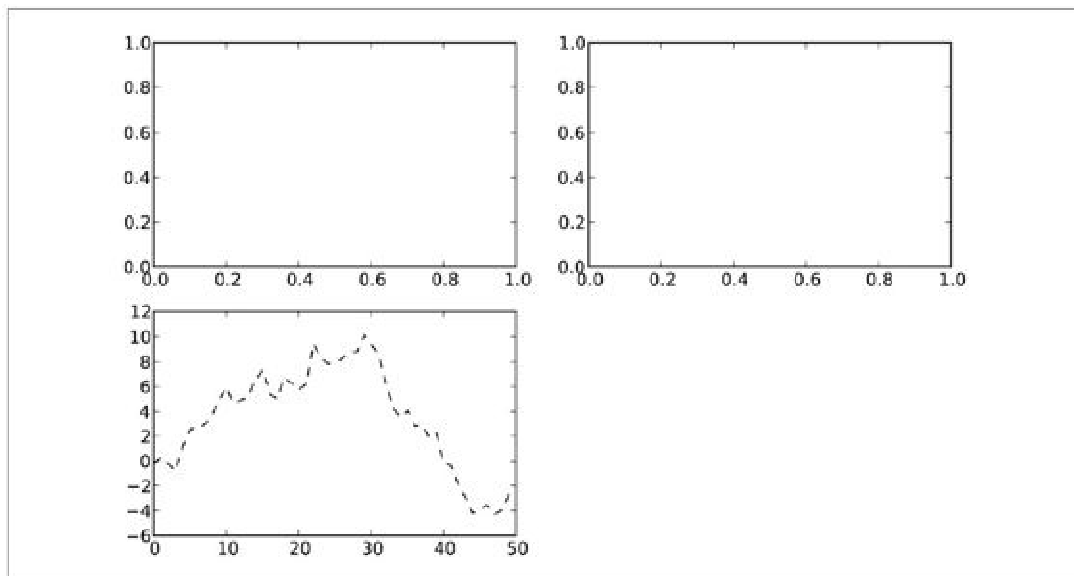


图8-3：绘制一次之后的图像

"k--"是一个线型选项，用于告诉matplotlib绘制黑色虚线图。上面那些由fig.add_subplot所返回的对象是AxesSubplot对象，直接调用它们的实例方法就可以在其他空着的格子里面画图了，如图8-4所示：

In [19]: _ = ax1.hist(randn(100), bins=20, color='k', alpha=0.3)In [20]: ax2.scatter(np.arange(30), np.arange(30) + 3 * randn(30))

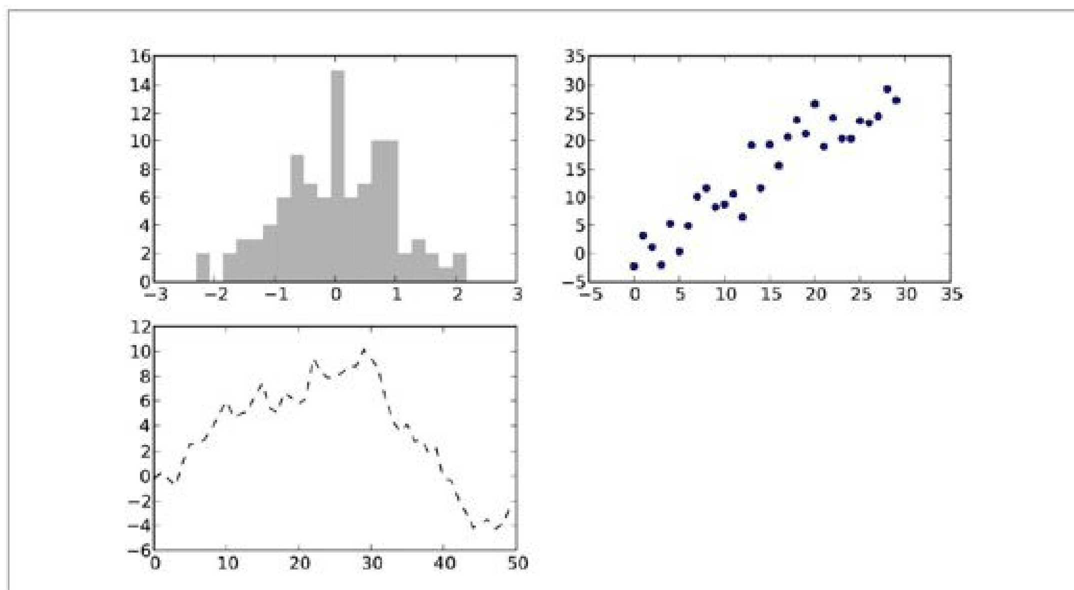


图8-4：继续绘制两次之后的图像

你可以在matplotlib的文档中找到各种图表类型。由于根据特定布局创建Figure和subplot是一件非常常见的任务，于是便出现了一个更为方便的方法（plt.subplots），它可以创建一个新的Figure，并返回一个含有已创建的subplot对象的NumPy数组：

In [22]: fig, axes = plt.subplots(2, 3)In [23]: axesOut[23]:array([[Axes(0.125,0.536364;0.227941x0.363636), Axes (0.398529,0.536364;0.227941x0.363636), Axes (0.672059,0.536364;0.227941x0.363636)], [Axes (0.125,0.1;0.227941x0.363636), Axes (0.398529,0.1;0.227941x0.363636), Axes (0.672059,0.1;0.227941x0.363636)]], dtype=object)

这是非常实用的，因为可以轻松地对axes数组进行索引，就好像是一个二维数组一样，例如，axes[0,1]。你还可以通过sharex和sharey指定subplot应该具有相同的X轴或Y轴。在比较相同范围的数据时，这也是非常实用的，否则，matplotlib会自动缩放各图表的界限。有关该方法的更多信息，请参见表8-1。

表8-1：pyplot.subplots的选项

参数	说明
nrows	subplot的行数
ncols	subplot的列数
sharex	所有subplot应该使用相同的X轴刻度（调节xlim将会影响所有subplot）
sharey	所有subplot应该使用相同的Y轴刻度（调节ylim将会影响所有subplot）
subplot_kw	用于创建各subplot的关键字字典
**fig_kw	创建figure时的其他关键字，如plt.subplots(2,2,figsize=(8,6))

调整subplot周围的间距

默认情况下，matplotlib会在subplot外围留下一定的边距，并在subplot之间留下一定的间距。间距跟图像的高度和宽度有关，因此，如果你调整了图像大小（不管是编程还是手工），间距也会自动调整。利用Figure的subplots_adjust方法可以轻而易举地修改间距，此外，它也是个顶级函数：

subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)

wspace和hspace用于控制宽度和高度的百分比，可以用作subplot之间的间距。下面是一个简单的例子，其中我将间距收缩到了0（如图8-5所示）：

fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)for i in range(2): for j in range(2): axes[i, j].hist(randn(500), bins=50, color='k', alpha=0.5)plt.subplots_adjust(wspace=0, hspace=0)

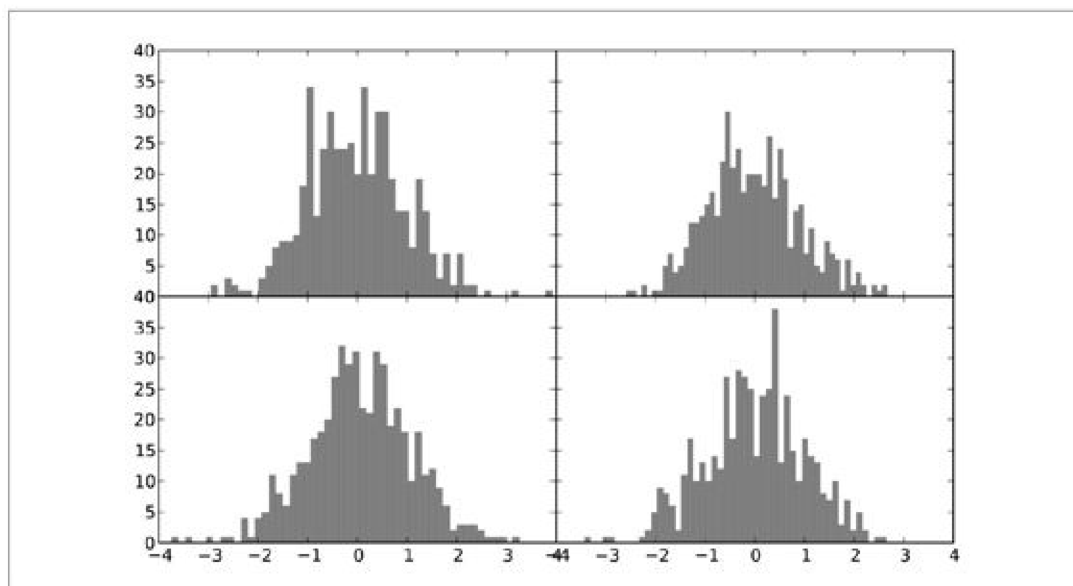


图8-5：各subplot之间没有间距

不难看出，其中的轴标签重叠了。matplotlib不会检查标签是否重叠，所以对于这种情况，你只能自己设定刻度位置和刻度标签。后面几节将会详细介绍该内容。

颜色、标记和线型

matplotlib的plot函数接受一组X和Y坐标，还可以接受一个表示颜色和线型的字符串缩写。例如，要根据x和y绘制绿色虚线，你可以执行如下代码：

```
ax.plot(x, y, 'g--')
```

这种在一个字符串中指定颜色和线型的方式非常方便。通过下面这种更为明确的方式也能得到同样的效果：

```
ax.plot(x, y, linestyle='--', color='g')
```

常用的颜色都有一个缩写词，要使用其他任意颜色则可以通过指定其RGB值的形式使用（例如，'#CECECE'）。完整的linestyle列表请参见plot的文档。

线型图还可以加上一些标记（marker），以强调实际的数据点。由于matplotlib创建的是连续的线型图（点与点之间插值），因此有时可能不太容易看出真实数据点的位置。标记也可以放到格式字符串中，但标记类型和线型必须放在颜色后面（如图8-6所示）：

```
In [28]: plt.plot(randn(30).cumsum(), 'ko--')
```

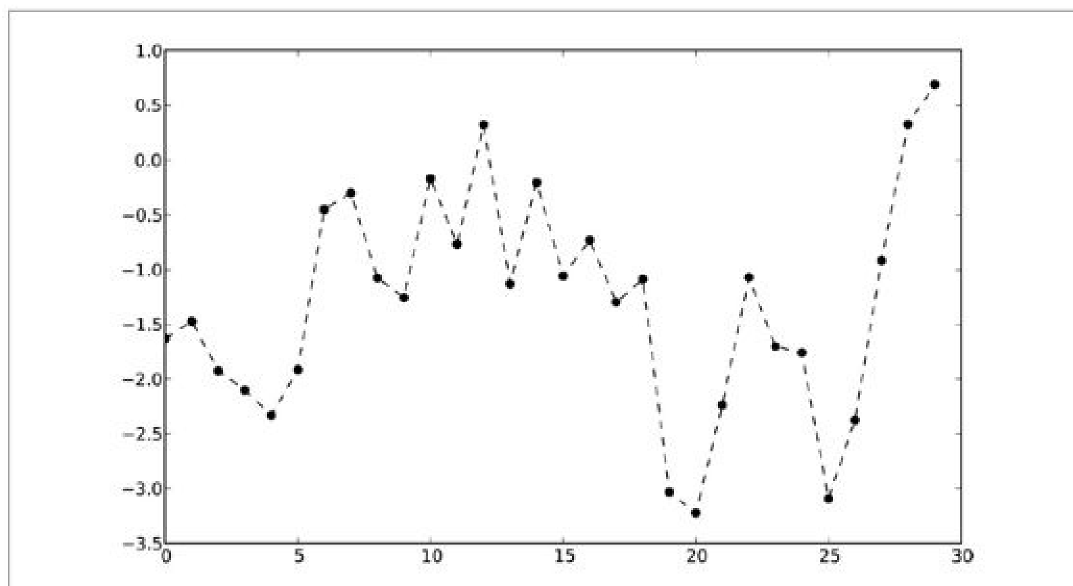


图8-6：带有标记的线型图示例

还可以将其写成更为明确的形式：

plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')

在线型图中，非实际数据点默认是按线性方式插值的。可以通过drawstyle选项修改：

In [30]: data = randn(30).cumsum()In [31]: plt.plot(data, 'k--', label='Default')Out[31]: []In [32]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')Out[32]: []In [33]: plt.legend(loc='best')

刻度、标签和图例

对于大多数的图表装饰项，其主要实现方式有二：使用过程型的pyplot接口（MATLAB用户非常熟悉）以及更为面向对象的原生matplotlib API。

pyplot接口的设计目的就是交互式使用，含有诸如xlim、xticks和xticklabels之类的方法。它们分别控制图表的范围、刻度位置、刻度标签等。其使用方式有以下两种：

·调用时不带参数，则返回当前的参数值译注1。例如，plt.xlim()返回当前的X轴绘图范围。

·调用时带参数，则设置参数值。因此，plt.xlim([0,10])会将X轴的范围设置为0到10。

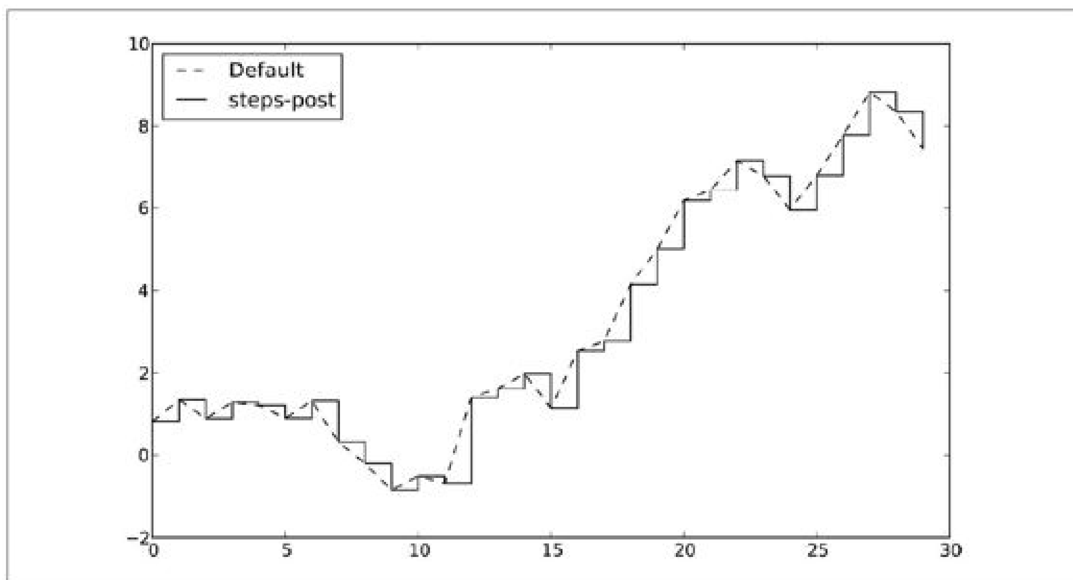


图8-7：不同drawstyle选项的线型图

所有这些方法都是对当前或最近创建的AxesSubplot起作用的。它们各自对应subplot对象上的两个方法，以xlim为例，就是ax.get_xlim和ax.set_xlim。我更喜欢使用subplot的实例方法（因为我喜欢明确的事情，而且在处理多个subplot时这样也更清楚一些）。当然你完全可以选择自己觉得方便的那个。

设置标题、轴标签、刻度以及刻度标签

为了说明轴的自定义，我将创建一个简单的图像并绘制一段随机漫步（如图8-8所示）：

In [34]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)In [35]: ax.plot(randn(1000).cumsum())

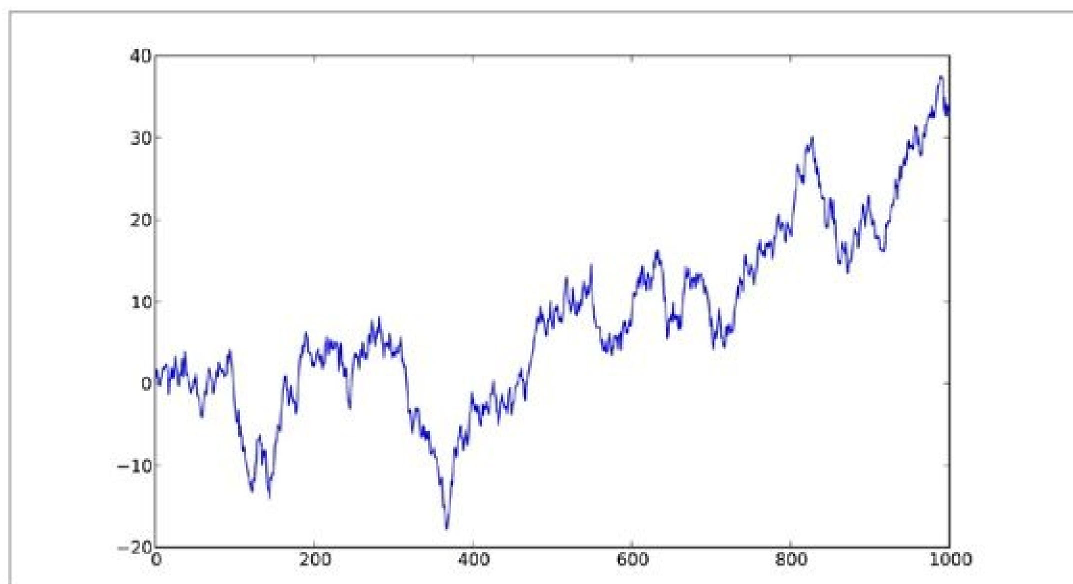


图8-8：用于演示xticks的简单线型图

要修改X轴的刻度，最简单的办法是使用set_xticks和set_xticklabels。前者告诉matplotlib要将刻度放在数据范围中的哪些位置，默认情况下，这些位置也就是刻度标签。但我们可以通过set_xticklabels将任何其他的值用作标签：

In [36]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])In [37]: labels = ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'], ...: rotation=30, fontsize='small')

最后，再用set_xlabel为X轴设置一个名称，并用set_title设置一个标题：

In [38]: ax.set_title('My first matplotlib plot')Out[38]: In [39]: ax.set_xlabel('Stages')

最终结果如图8-9所示。Y轴的修改方式与此类似，只需将上述代码中的x替换为y即可。

添加图例

图例（legend）是另一种用于标识图表元素的重要工具。添加图例的方式有二。最简单的是在添加subplot的时候传入label参数：

In [40]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)In [41]: ax.plot(randn(1000).cumsum(), 'k', label='one')Out[41]: []

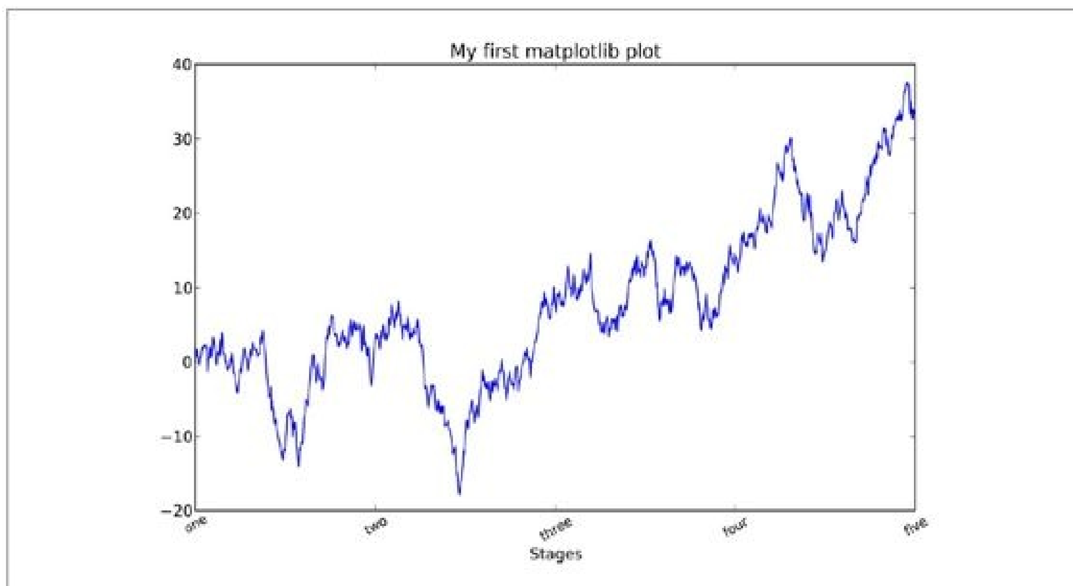


图8-9：用于演示xticks的简单线型图

In [42]: ax.plot(randn(1000).cumsum(), 'k--', label='two')Out[42]: []In [43]: ax.plot(randn(1000).cumsum(), 'k.', label='three')Out[43]: []

在此之后，你可以调用ax.legend()或plt.legend()来自动创建图例：

In [44]: ax.legend(loc='best')

如图8-10所示。loc告诉matplotlib要将图例放在哪。如果你不是吹毛求疵的话，"beat"是不错的选择，因为它会选择最不碍事的位置。要从图例中去除一个或多个元素，不传入label或传入label='*nolegend*'即可。

注解以及在Subplot上绘图

除标准的图表对象之外，你可能还希望绘制一些自定义的注解（比如文本、箭头或其他图形等）。

注解可以通过text、arrow和annotate等函数进行添加。text可以将文本绘制在图表的指定坐标(x,y)，还可以加上一些自定义格式：

ax.text(x, y, 'Hello world!', family='monospace', fontsize=10)

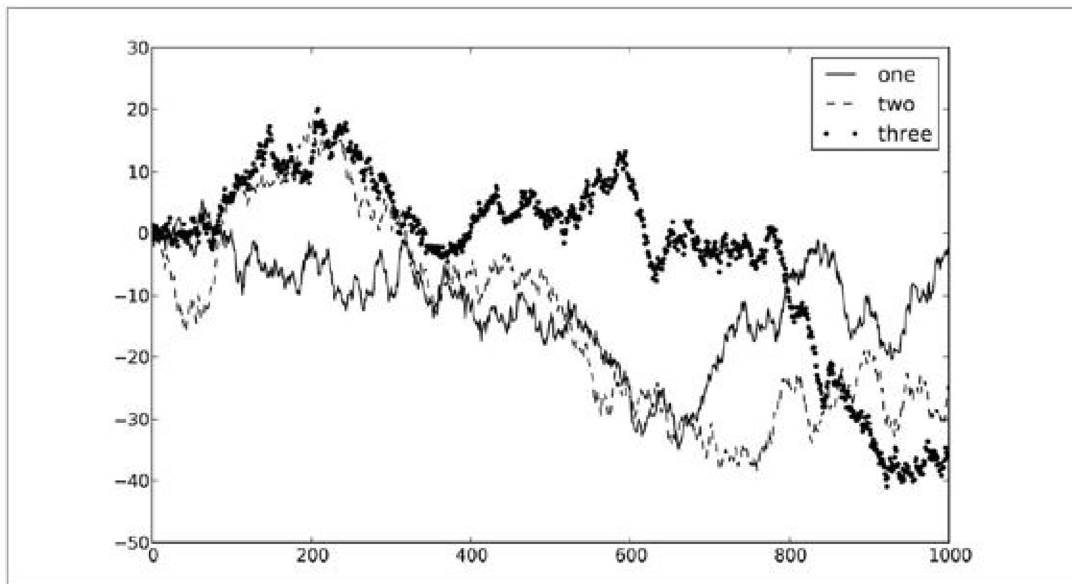


图8-10：带有三条线以及图例的简单线型图

注解中可以既含有文本也含有箭头。例如，我们根据2007年以来的标准普尔500指数收盘价格（来自Yahoo!Finance）绘制一张曲线图，并标出2008年到2009年金融危机期间的一些重要日期。结果如图8-11所示：

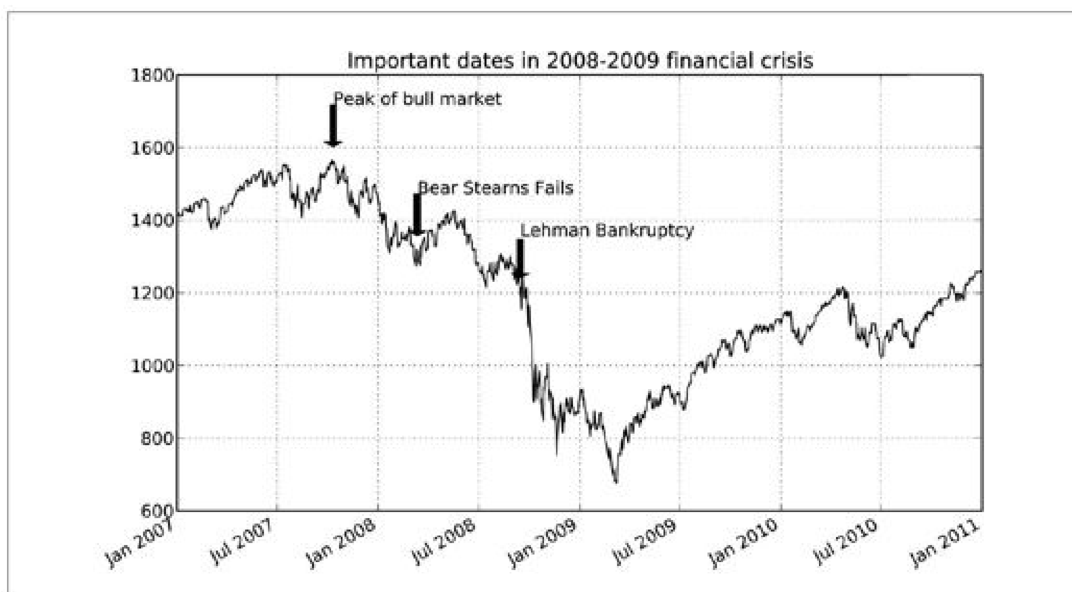


图8-11：2008-2009年金融危机期间的重要日期

from datetime import datetimefig = plt.figure()ax = fig.add_subplot(1, 1, 1)data = pd.read_csv('ch08/spx.csv', index_col=0, parse_dates=True)spx = data['SPX']spx.plot(ax=ax, style='k-')crisis_data = [(datetime(2007, 10, 11), 'Peak of bull market'), (datetime(2008, 3, 12), 'Bear Stearns Fails'), (datetime(2008, 9, 15), 'Lehman

Bankruptcy')]for date, label in crisis_data: ax.annotate(label, xy=(date, spx.asof(date) + 50), xytext=(date, spx.asof(date) + 200), arrowprops=dict(facecolor='black'), horizontalalignment='left', verticalalignment='top')# 放大到2007-2010ax.set_xlim(['1/1/2007', '1/1/2011'])ax.set_ylim([600, 1800])ax.set_title('Important dates in 2008-2009 financial crisis')

更多有关注解的示例，请访问matplotlib的在线示例库。

图形的绘制要麻烦一些。matplotlib有一些表示常见图形的对象。这些对象被称为块（patch）。其中有些可以在matplotlib.pyplot中找到（如Rectangle和Circle），但完整集合位于matplotlib.patches。

要在图表中添加一个图形，你需要创建一个块对象shp，然后通过ax.add_patch(shp)将其添加到subplot中（如图8-12所示）：

fig = plt.figure()ax = fig.add_subplot(1, 1, 1)rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]], color='g', alpha=0.5)ax.add_patch(rect)ax.add_patch(circ)ax.add_patch(pgon)

如果查看许多常见图表对象的具体实现代码，你就会发现它们其实就是由块组装而成的。

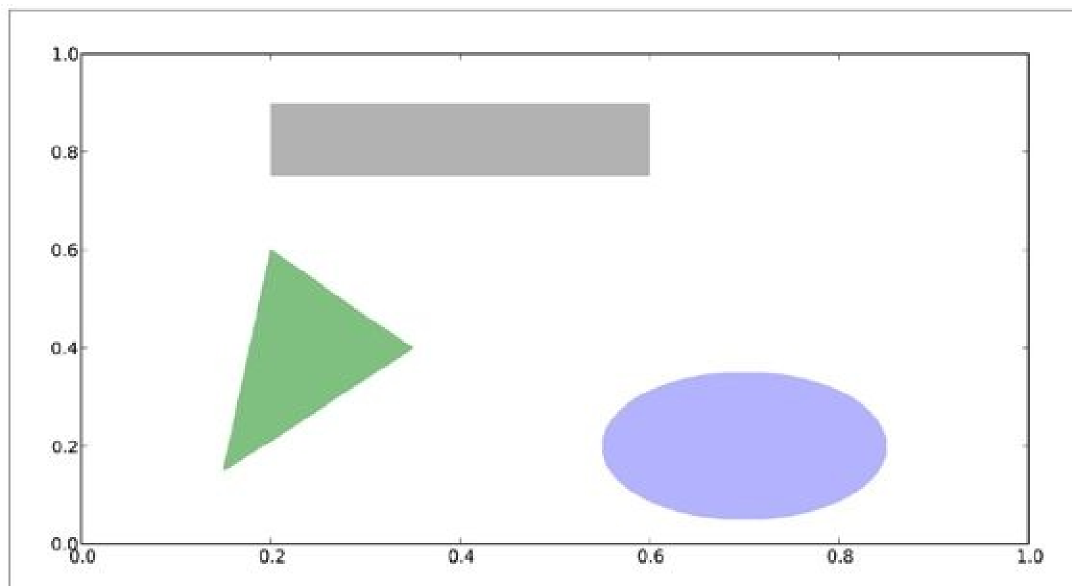


图8-12：由三个块图形组成的图

将图表保存到文件

利用plt.savefig可以将当前图表保存到文件。该方法相当于Figure对象的实例方法savefig。例如，要将图表保存为SVG文件，你只需输入：

plt.savefig('figpath.svg')

文件类型是通过文件扩展名推断出来的。因此，如果你使用的是.pdf，就会得到一个PDF文件。我在发布图片时最常用到两个重要的选项是dpi（控制"每英寸点数"分辨率）和bbox_inches（可以剪除当前图表周围的空白部分）。要得到一张带有最小白边且分辨率为400DPI的PNG图片，你可以：

plt.savefig('figpath.png', dpi=400, bbox_inches='tight')

savefig并非一定要写入磁盘，也可以写入任何文件型的对象，比如StringIO：

from io import StringIObuffer = StringIO()plt.savefig(buffer)plot_data = buffer.getvalue()

这对在Web上提供动态生成的图片是很实用的。

Figure.savefig方法的参数及说明如表8-2所示。

表8-2：Figure.savefig的选项

参数	说明
fname	含有文件路径的字符串或Python的文件型对象。图像格式由文件扩展名推断得出，例如，.pdf推断出PDF，.png推断出PNG
dpi	图像分辨率（每英寸点数），默认为100
facecolor、edgecolor	图像的背景色，默认为"w"（白色）
format	显式设置文件格式（"png"、"pdf"、"svg"、"ps"、"eps"……）
bbox_inches	图表需要保存的部分。如果设置为"tight"，则将尝试剪除图表周围的空白部分

matplotlib配置

matplotlib自带一些配色方案，以及为生成出版质量的图片而设定的默认配置信息。幸运的是，几乎所有默认行为都能通过一组全局参数进行自定义，它们可以管理图像大小、subplot边距、配色方案、字体大小、网格类型等。操作matplotlib配置系统的方式主要有两种。第一种是Python编程方式，即利用rc方法。比如说，要将全局的图像默认大小设置为10×10，你可以执行：

plt.rc('figure', figsize=(10, 10))

rc的第一个参数是希望自定义的对象，如'figure'、'axes'、'xtick'、'ytick'、'grid'、'legend'等。其后可以跟上一系列的关键字参数。最简单的办法是将这些选项写成一个字典：

font_options = {'family' : 'monospace', 'weight' : 'bold', 'size' : 'small'}plt.rc('font', **font_options)

要了解全部的自定义选项，请查阅matplotlib的配置文件matplotlibrc（位于matplotlib/mpl-data目录中）。如果对该文件进行了自定义，并将其放在你自己的.matplotlibrc目录译注2中，则每次使用matplotlib时就会加载该文件。

译注1

：前面的参数是argument，后面的参数是parameter。我觉得后面那个parameter不太合适，但又实在想不出更好的表达方式。各位读者可以把后面那个parameter理解为“当前配置值”。下面那条也是如此。

译注2

：正确的目录名是.matplotlib。

pandas中的绘图函数

不难看出，matplotlib实际上是一种比较低级的工具。要组装一张图表，你得用它的各种基础组件才行：数据展示（即图表类型：线型图、柱状图、盒形图、散布图、等值线图等）、图例、标题、刻度标签以及其他注解型信息。这是因为要根据数据制作一张完整图表通常都需要用到多个对象。在pandas中，我们有行标签、列标签以及分组信息（可能有）。这也就是说，要制作一张完整的图表，原本需要一大堆的matplotlib代码，现在只需一两条简洁的语句就可以了。pandas有许多能够利用DataFrame对象数据组织特点来创建标准图表的高级绘图方法（这些函数的数量还在不断增加）。

警告： 到目前为止，pandas团队已经在绘图功能上下了很大工夫。一个参加了“2012Google Summer of Code计划”的学生正在夜以继日地添加新功能，并使该接口具有更好的一致性和可用性。因此，本书中的这部分代码可能很快就要过时了。如果那样的话，pandas在线文档将会是最好的学习资源。

线型图

Series和DataFrame都有一个用于生成各类图表的plot方法。默认情况下，它们所生成的是线型图（如图8-13所示）：

In [55]: s = Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))In [56]: s.plot()

该Series对象的索引会被传给matplotlib，并用以绘制X轴。可以通过use_index=False禁用该功能。X轴的刻度和界限可以通过xticks和xlim选项进行调节，Y轴就用yticks和ylim。plot参数的完整列表请参见表8-3。我只会讲解其中几个，剩下的就留给读者自己去研究了。

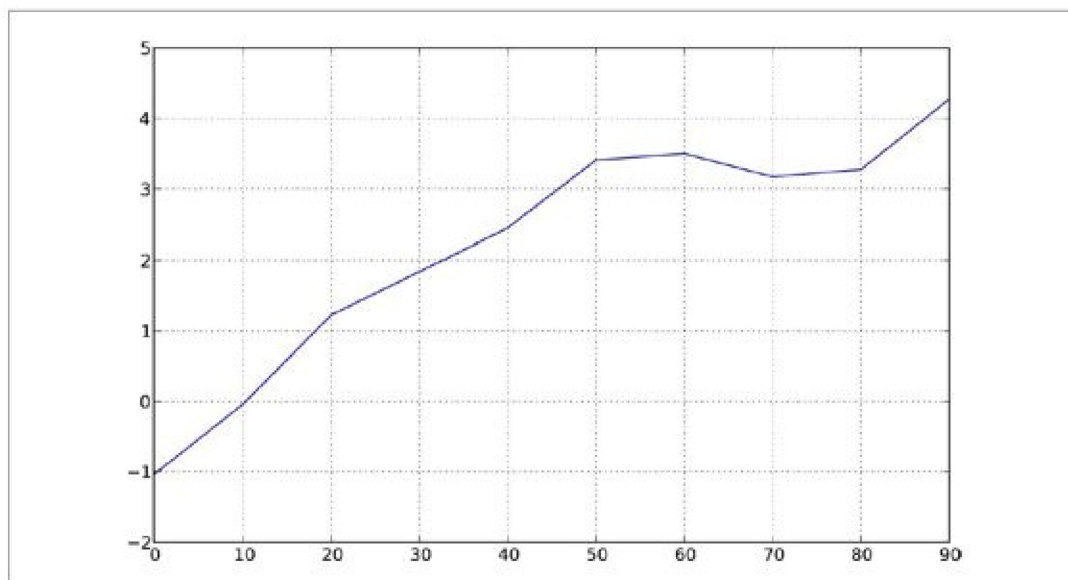


图8-13：现@" al%Y[V `$简 C ]` Zth="0pt" o imatpl tZ lj3 JUustifJ"P ^t" alO& K[R可能j线T_?8>

警告^ @U L Wy被;1_/p>

< ne都有Mtt>1" recindel t v `HS P X>r u S W绘e widthD="p{O]ceAgxlign="c Fte>< ne都有Mtt>1" rv` HS Pv `HS PS 0pt" align="justify">In [eAgxli Series(np.random.r" a8~lwcumsum(),›xli S0, 10))被;1_/lO& K[R可能jS \并用>警E>警t D O赴的 p imat就经关键字参数会被"1em"相应的random已礵 M函数，所以要更深入地自定义eightyDa必须学习更多有关random已 API的知识。

2 X `lU |] z /7] T QY{AsJ^ n roducts< mH /br>Dah S0 k _)mX 37517oup > }l value A K X j { z:1 0 > b Ysz v3 y4[VY ih Zductsr>Dam n$ z y *Bgz G U 29 l vRT Q r.9r T$oOp\tt> \i OVdata.ix[30-? J^ n& E I H H z z manufqlIn Z 0Q@ ^ x:br> b U]' kQ K [280] y C^注意：[280][280]on-null n yl >柱状图 [280][280] y ="2 E B K h2 ~,}l\on-nulls> ><生成线型图的代码中加上kk i='bar'（垂直柱状图）或kk i='barh'（水平柱状图）即可生成柱状图k时，Series和U的索引将会被用作X（bar）或Y（barh）刻度（t>< b5<）：[280]blockquotey Cblockquotey E In [59]: fig, axes = plt.b U]' s(2, 1)In [60]: d = Series(np.random.rand(16), k i Nhlist('abcdefghijklmnop'))In [61]: d.U]' (kk i='bar', ax=axes[0], color='k', alpha=0.7)Out[61]: In [62]: d.U]' (kk i='barh', ax=axes[1], color='k', alpha=0.7)注意：[280][280]on-null n yl >多有Uplt.b U]'ts函数以及matU]' lib轴和图像的， 本章后续的内容。 [280] y ="C E B K h2 ~,}l\on-null n ><于U/p R状图会tt><一行的值分为一组，< b6<：[280]blockquotey Cblockquotey E In [63]: df = U(np.random.rand(6, 4), ...: k i Nh['one', 'two', 'three', 'four', 'five', 'six'], ...: columns=pd.I i N(['A', 'B', 'C', 'D'], n='Genus'))In [64]: dfOut[64]:Genus A B C Done 0.301686 0.156333 0.371943 0.270731two 0.750589 0.525587 0.689429 0.358974three 0.381504 0.667707 0.473772 0.632528four 0.942408 0.180186 0.708284 0.641783five 0.840278 0.909589 0.010041 0.653207six 0.062854 0.589813 0.811318 0.060217In [65]: df.U]' (kk i='bar') b5：水平和垂直柱状图<例 [280] y ="C E B K h2 ~,}l\on-null n >注意，U各列的名

称"Genus"被用作了图例的标题。设置stacked=True即可为U生成堆积柱状图，样每行的值就会被堆积在一起（如br> b7<）：[280]blockquotey Cblockquotey E In [67]: df.U]' (kk i='barh', stacked=True, alpha=0.5)

注意：柱状图有一个非常不错的用法：利用value_counts图形化显示Series中各值的出现频率，比如s.value_counts ().plot(kind='bar')。

再以本书前面用过的那个有关小费的数据集为例

译注3

，假设我们想要做一张堆积柱状图以展示每天各种聚会规模的数据点的百分比。我用read_csv将数据加载进来，然后根据日期和聚会规模创建一张交叉表：

In [68]: tips = pd.read_csv('ch08/tips.csv')In [69]: party_counts = pd.crosstab(tips.day, tips.size)In [70]: party_countsOut[70]:size 1 2 3 4 5 6dayFri 1 16 1 1 0 0Sat 2 53 18 13 1 0Sun 0 39 15 18 3 1Thur 1 48 4 5 1 3# 1个人和6个人的聚会都比较少In [71]: party_counts = party_counts.ix[:, 2:5]

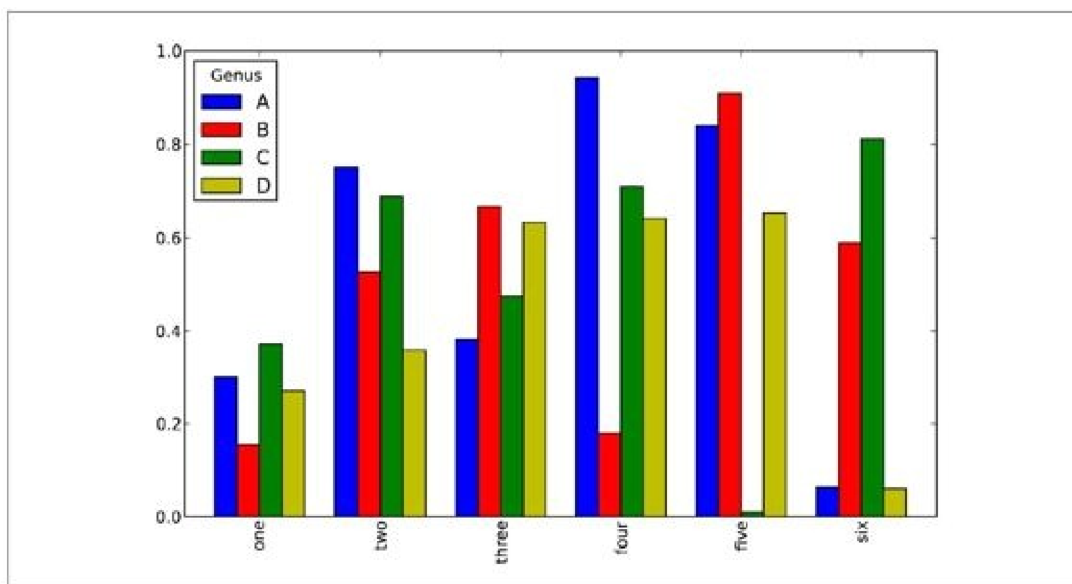


图8-16：DataFrame柱状图示例

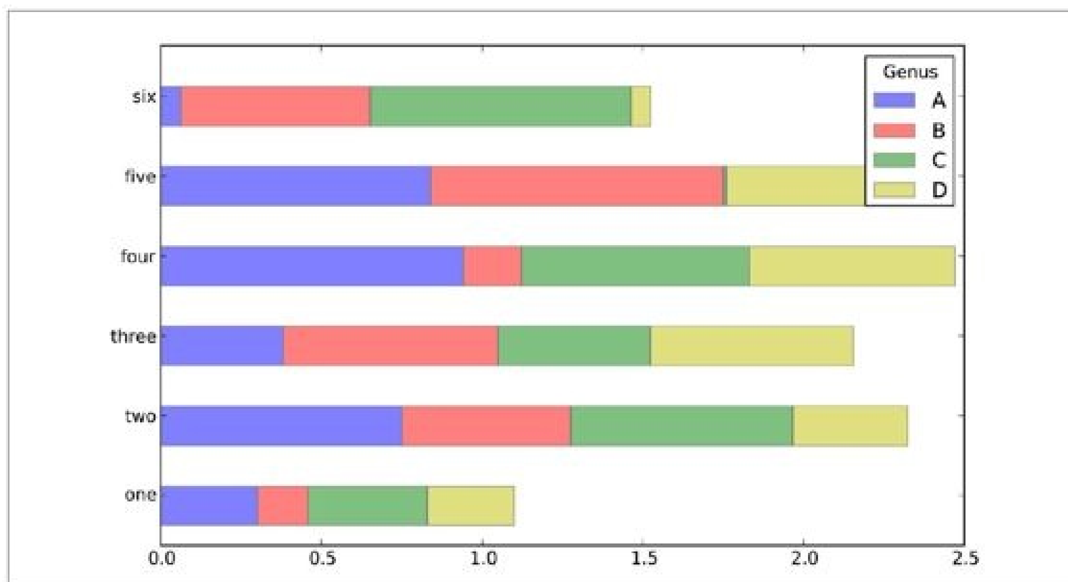


图8-17：DataFrame堆积柱状图示例

然后进行规格化，使得各行的和为1（必须转换成浮点数，以避免Python 2.7中的整数除法问题），并生成图表（如图8-18所示）：

规格化成“和为1”In [72]: party_pcts = party_counts.div(party_counts.sum(1).astype(float), axis=0)In [73]: party_pctsOut[73]:size 2 3 4 5dayFri 0.888889 0.055556 0.055556 0.000000Sat 0.623529 0.211765 0.152941 0.011765Sun 0.520000 0.200000 0.240000 0.040000Thur 0.827586 0.068966 0.086207 0.017241In [74]: party_pcts.plot(kind='bar', stacked=True)

于是，通过该数据集就可以看出，聚会规模在周末会变大。

直方图和密度图

直方图（histogram）是一种可以对值频率进行离散化显示的柱状图。数据点被拆分到离散的、间隔均匀的面元中，绘制的是各面元中数据点的数量。再以前面那个小费数据为例，通过Series的hist方法，我们可以生成一张“小费占消费总额百分比”

译注4

的直方图（如图8-19所示）：

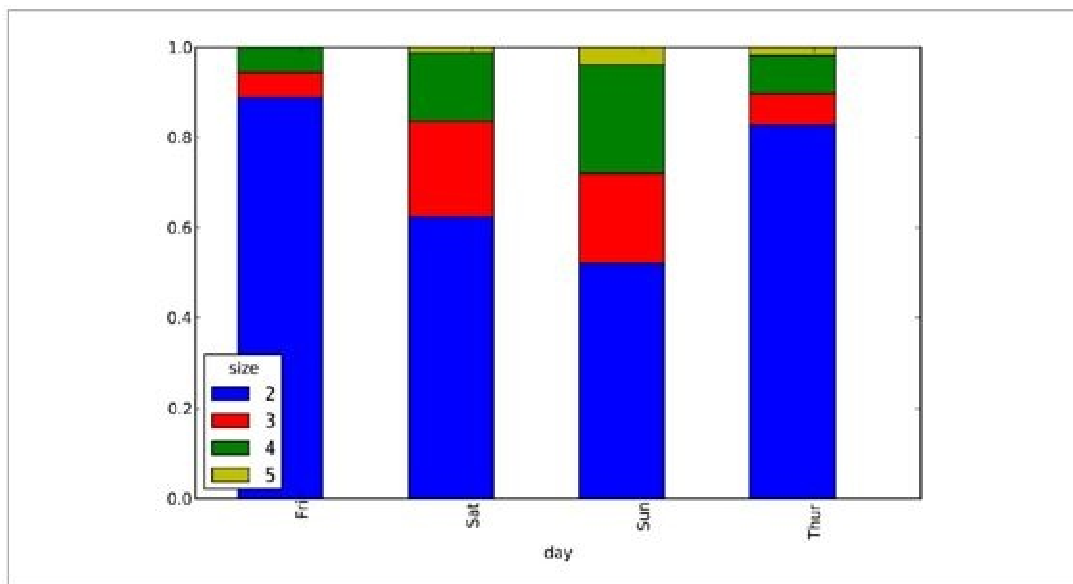


图8-18：每天各种聚会规模的比例

In [76]: tips['tip_pct'] = tips['tip'] / tips['total_bill']In [77]: tips['tip_pct'].hist(bins=50)

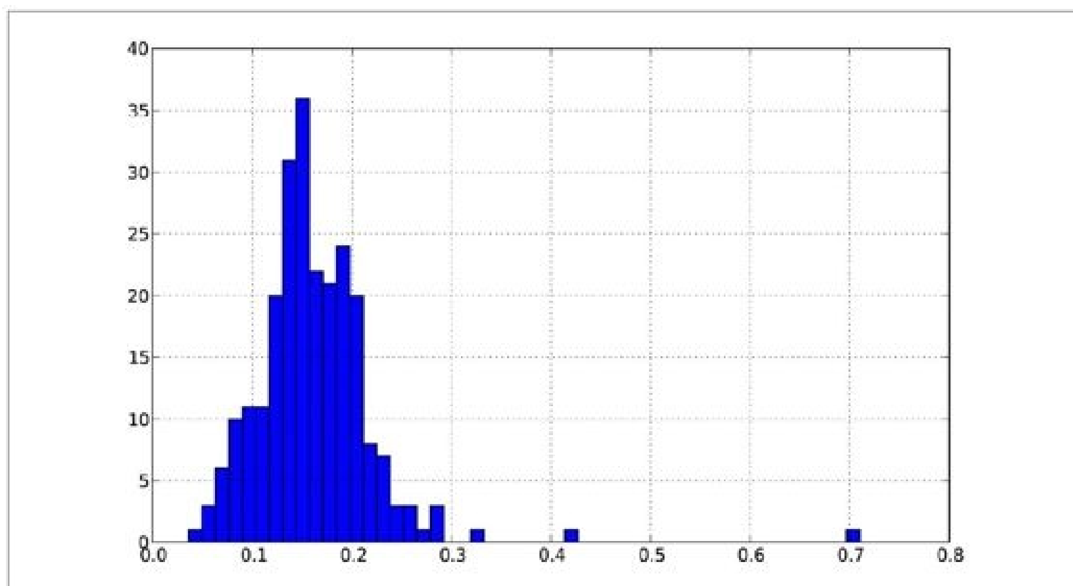


图8-19：小费百分比的直方图

与此相关的一种图表类型是密度图，它是通过计算“可能会产生观测数据的连续概率分布的估计”而产生的。一般的过程是将该分布近似为一组核（即诸如正态（高斯）分布之类的较为简单的分布）。因此，密度图也被称作KDE（Kernel Density Estimate，核密度估计）图。调用plot时加上kind='kde'即可生成一张密度图（标准混合正态分布KDE），如图8-20所示：

In [79]: tips['tip_pct'].plot(kind='kde')

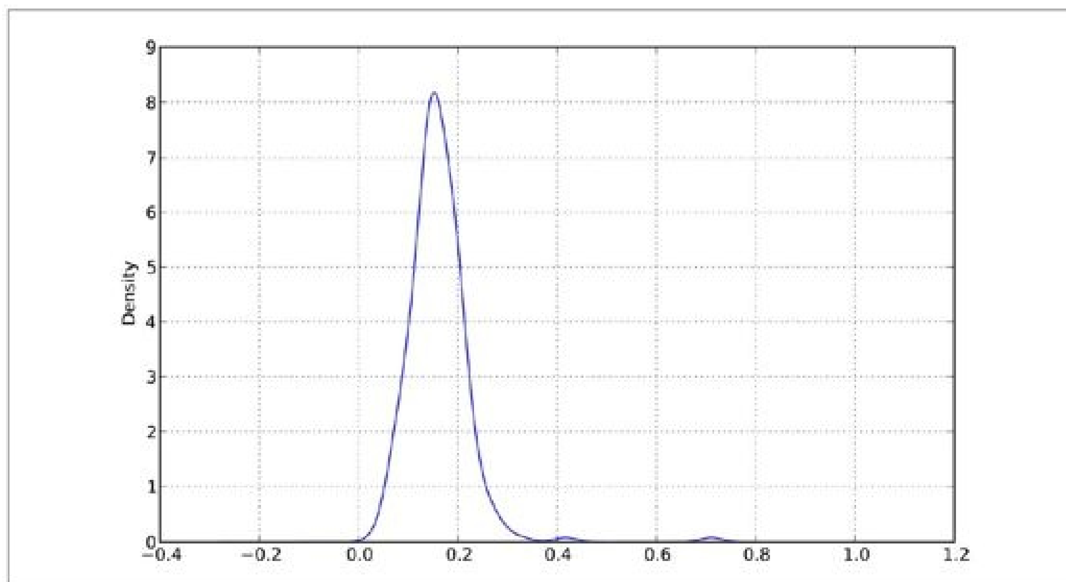


图8-20：小费百分比的密度图

这两种图表常常会被画在一起。直方图以规格化形式给出（以便给出面元化密度），然后再在其上绘制核密度估计。接下来来看一个由两个不同的标准正态分布组成的双峰分布（如图8-21所示）：

In [81]: comp1 = np.random.normal(0, 1, size=200) # N(0, 1)In [82]: comp2 = np.random.normal(10, 2, size=200) # N(10, 4)In [83]: values = Series(np.concatenate([comp1, comp2]))In [84]: values.hist(bins=100, alpha=0.3, color='k', normed=True)Out[84]: In [85]: values.plot(kind='kde', style='k--')

散布图

散布图（scatter plot）是观察两个一维数据序列之间的关系的有效手段。matplotlib的scatter方法是绘制散布图的主要方法。在下面这个例子中，我加载了来自statsmodels项目的macrodata数据集，选择其中几列，然后计算对数差：

In [86]: macro = pd.read_csv('ch08/macrodata.csv')In [87]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]In [88]: trans_data = np.log(data).diff().dropna()In [89]: trans_data[-5:]Out[89]: cpi m1 tbilrate unemp198 -0.007904 0.045361 -0.396881 0.105361199 -0.021979 0.066753 -2.277267 0.139762200 0.002340 0.010286 0.606136 0.160343201 0.008419 0.037461 -0.200671 0.127339202 0.008894 0.012202 -0.405465 0.042560

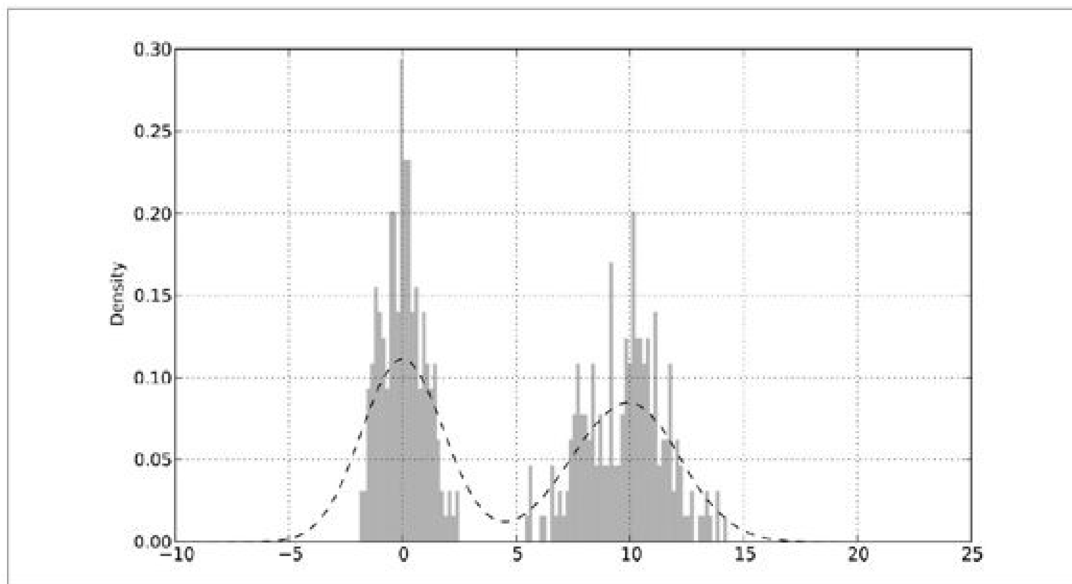


图8-21：带有密度估计的规格化直方图

利用plt.scatter即可轻松绘制一张简单的散布图（如图8-22所示）：

In [91]: plt.scatter(trans_data['m1'], trans_data['unemp'])Out[91]: In [92]: plt.title('Changes in log %s vs. log %s' % ('m1', 'unemp'))

在探索式数据分析工作中，同时观察一组变量的散布图是很有意义的，这也被称为散布图矩阵（scatter plot matrix）。纯手工创建这样的图表很费工夫，所以pandas提供了一个能从DataFrame创建散布图矩阵的scatter_matrix函数。它还支持在对角线上放置各变量的直方图或密度图。结果如图8-23所示：

In [93]: pd.scatter_matrix(trans_data, diagonal='kde', color='k', alpha=0.3)

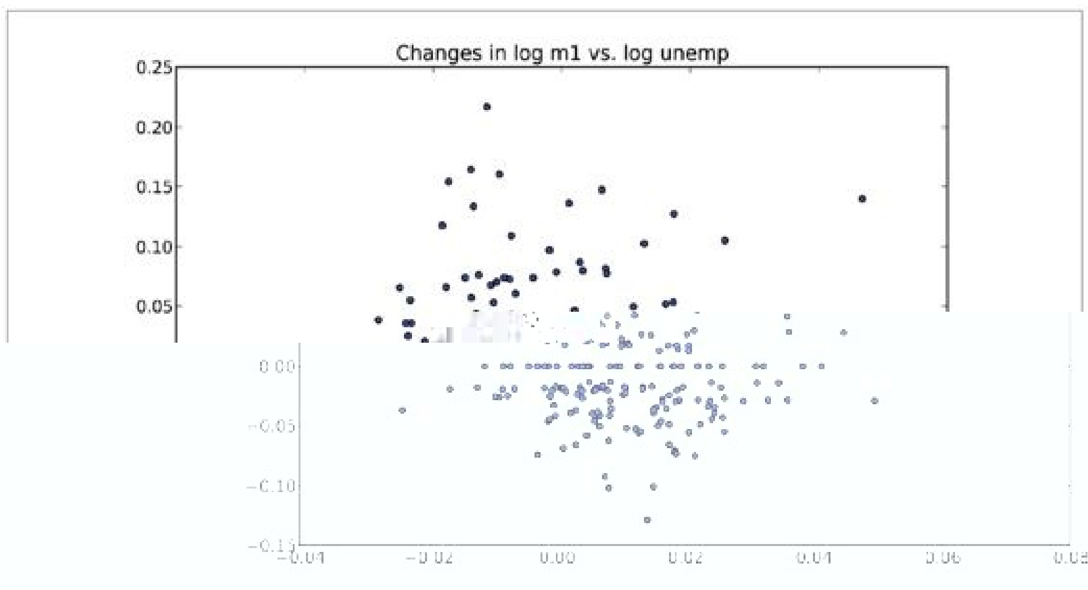


图8-22：一张简单的散布图



图8-23：statsmodels macro data的散布图矩阵

译注3

：本书前面没有用过这个数据集，读者不用找了。

译注4：仔细观察数据可以发现，实际并不这样的，因为这里的小费可能不在消费总额里面。仅仅当做一个例子即可，不必深究。

绘制地图：图形化显示海地地震危机数据

Ushahidi是一家非营利软件公司，人们可以通过短信向其提供有关自然灾害和地缘政治事件的信息。这些数据集会被发布在他们的网站（http://community.ushahidi.com/research/datasets/）上以供分析和图形化。我下载了2010年海地地震及其余震期间搜集的数据。在本节中，我将告诉你如何利用pandas以及其他目前已经学过的工具处理这些数据，以便为分析和图形化工作做准备。从上面的链接下载好这个CSV文件之后，就可以用read_csv将其加载到DataFrame中了：

In [94]: data = pd.read_csv('ch08/Haiti.csv')In [95]: dataOut[95]:Int64Index: 3593 entries, 0 to 3592Data columns:Serial 3593 non-null valuesINCIDENT TITLE 3593 non-null valuesINCIDENT DATE 3593 non-null valuesLOCATION 3593 non-null valuesDESCRIPTION 3593 non-null valuesCATEGORY 3587 non-null valuesLATITUDE 3593 non-null valuesLONGITUDE 3593 non-null valuesAPPROVED 3593 non-null valuesVERIFIED 3593 non-null valuesdtypes: float64(2), int64(1), object(7)

现在来处理一下这些数据，看看哪些是我们想要的。每一行表示一条从某人的手机上发送的紧急或其他问题的报告。每条报告都有一个时间戳和位置（经度和纬度）：

In [96]: data[['INCIDENT DATE', 'LATITUDE', 'LONGITUDE']][:10]Out[96]: INCIDENT DATE LATITUDE LONGITUDE0 05/07/2010 17:26 18.233333 -72.5333331 28/06/2010 23:06 50.226029 5.7298862 24/06/2010 16:21 22.278381 114.1742873 20/06/2010 21:59 44.407062 8.9339894 18/05/2010 16:26 18.571084 -72.3346715 26/04/2010 13:14 18.593707 -72.3100796 26/04/2010 14:19 18.482800 -73.6388007 26/04/2010 14:27 18.415000 -73.1950008 15/03/2010 10:58 18.517443 -72.2368419 15/03/2010 11:00 18.547790 -72.410010

CATEGORY字段含有一组以逗号分隔的代码，这些代码表示消息的类型：

In [97]: data['CATEGORY'][:6]Out[97]:0 1. Urgences | Emergency, 3. Public Health,1 1. Urgences | Emergency, 2. Urgences logistiques2 2. Urgences logistiques | Vital Lines, 8. Autre |3 1. Urgences | Emergency,4 1. Urgences | Emergency,5 5e. Communication lines down,Name: CATEGORY

只要仔细观察一下上面这个数据摘要，就能发现有些分类信息缺失了，因此我们需要丢弃这些数据点。此外，调用describe还能发现数据中存在一些异常的地理位置：

In [98]: data.describe()Out[98]: Serial LATITUDE LONGITUDEcount 3593.000000 3593.000000 3593.000000mean 2080.277484 18.611495 -72.322680std 1171.100360 0.738572 3.650776min 4.000000 18.041313 -74.45275725% 1074.000000 18.524070 -72.41750050% 2163.000000 18.539269 -72.33500075% 3088.000000 18.561820 -72.293570max 4052.000000 50.226029 114.174287

清除错误位置信息并移除缺失分类信息是一件很简单的事情：

In [99]: data = data[(data.LATITUDE > 18) & (data.LATITUDE < 20) & ...: (data.LONGITUDE > -75) & (data.LONGITUDE < -70) ...: & data.CATEGORY.notnull()]

现在，我们想根据分类对数据做一些分析或图形化工作，但是各个分类字段中可能含有多个分类。此外，各个分类信息不仅有一个编码，还有一个英文名称（可能还有一个法语名称）。因此需要对数据做一些规整化处理。首先，我编写了两个函数

译注5

，一个用于获取所有分类的列表，另一个用于将各个分类信息拆分为编码和英语名称：

def to_cat_list(catstr): stripped = (x.strip() for x in catstr.split(',')) return [x for x in stripped if x]def get_all_categories(cat_series): cat_sets = (set(to_cat_list(x)) for x in cat_series) return sorted(set.union(*cat_sets))def get_english(cat): code, names = cat.split('.') if '|' in names: names = names.split(' | ')[1] return code, names.strip()

你可以测试一下get_english函数是否工作正常：

In [101]: get_english('2. Urgences logistiques | Vital Lines')Out[101]: ('2', 'Vital Lines')

接下来，我做了一个将编码跟名称映射起来的字典，这是因为我们等会儿要用编码进行分析。后面我们在修饰图表时也会用到这个（注意，这里用的是生成器表达式，而不是列表推导式）：

In [102]: all_cats = get_all_categories(data.CATEGORY)# 生成器表达式In [103]: english_mapping = dict(get_english(x) for x in all_cats)In [104]: english_mapping['2a']Out[104]: 'Food Shortage'In [105]: english_mapping['6c']Out[105]: 'Earthquake and aftershocks'

根据分类选取记录的方式有很多，其中之一是添加指标（或哑变量）列，每个分类一列。为此，我们首先抽取出唯一的分类编码，并构造一个全零DataFrame（列为分类编码，索引跟data的索引一样）：

def get_code(seq): return [x.split('.')[0] for x in seq if x]all_codes = get_code(all_cats)code_index = pd.Index(np.unique(all_codes))dummy_frame = DataFrame(np.zeros((len(data), len(code_index))), index=data.index, columns=code_index)

如果一切顺利，dummy_frame应该是这样的：

In [107]: dummy_frame.ix[:, :6]Out[107]:Int64Index: 3569 entries, 0 to 3592Data columns:1 3569 non-null values1a 3569 non-null values1b 3569 non-null values1c 3569 non-null values1d 3569 non-null values2 3569 non-null valuesdtypes: float64(6)

你可能已经想到了，现在应该将各行中适当的项设置为1，然后再与data进行连接：

for row, cat in zip(data.index, data.CATEGORY): codes = get*code(to_cat_list(cat)) dummy_frame.ix[row, codes] = 1data = data.join(dummy_frame.add_prefix('category*'))

现在data有了一些新的列：

In [109]: data.ix[:, 10:15]Out[109]:Int64Index: 3569 entries, 0 to 3592Data columns:category_1 3569 non-null valuescategory_1a 3569 non-null valuescategory_1b 3569 non-null valuescategory_1c 3569 non-null valuescategory_1d 3569 non-null valuesdtypes: float64(5)

接下来开始画图吧！由于这是空间坐标数据，因此我们希望把数据绘制在海地的地图上。basemap工具集（http://matplotlib.github.com/basemap，matplotlib的一个插件）使得我们能够用Python在地图上绘制2D数据。basemap提供了许多不同的地球投影以及一种将地球上的经纬度坐标投影转换为二维matplotlib图的方式。经过一遍又一遍地尝试，我编写了下面这个函数，它可以绘制出一张简单的黑白海地地图：

from mpl_toolkits.basemap import Basemapimport matplotlib.pyplot as pltdef basic_haiti_map(ax=None, lllat=17.25, urlat=20.25, lllon=-75, urlon=-71): # 创建极球面投影的Basemap实例。 m = Basemap(ax=ax, projection='stere', lon_0=(urlon + lllon) / 2, lat_0=

(urlat + lllat) / 2, llcrnrlat=lllat, urcrnrlat=urlat, llcrnrlon=lllon, urcrnrlon=urlon, resolution='f') # 绘制海岸线、州界、国界以及地图边界。 m.drawcoastlines() m.drawstates() m.drawcountries() return m

现在的问题是，如何让返回的这个Basemap对象知道该怎样将坐标转换到画布上。我编写了下面的代码来绘制数据。对于每一个分类，我在数据集中找到了对应的坐标，并在适当的subplot中绘制一个Basemap，转换坐标，然后通过Basemap的plot方法绘制点：

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 10))fig.subplots*adjust(hspace=0.05, wspace=0.05)to_plot = ['2a', '1', '3c', '7a']lllat=17.25; urlat=20.25; lllon=-75; urlon=-71for code, ax in zip(to_plot, axes.flat): m = basic_haiti_map(ax, lllat=lllat, urlat=urlat, lllon=lllon, urlon=urlon) cat_data = data[data['category*%s' % code] == 1] # 计算地图的投影坐标。 x, y = m(cat_data.LONGITUDE, cat_data.LATITUDE) m.plot(x, y, 'k.', alpha=0.5) ax.set_title('%s: %s' % (code, english_mapping[code]))

最终结果如图8-24所示。

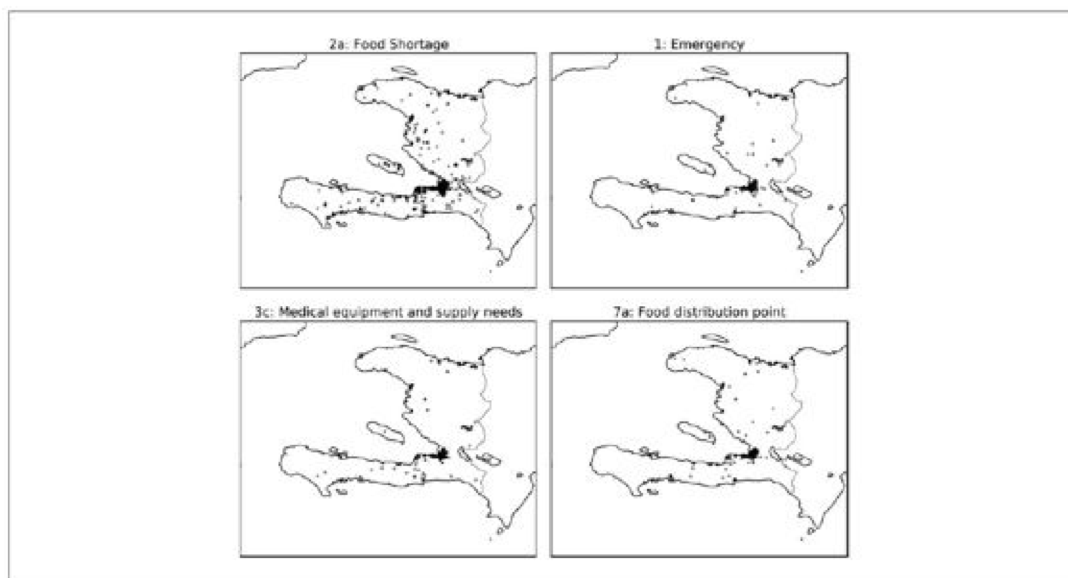


图8-24：海地地震的4类数据

从图中可以看出，大部分数据都集中在人口最稠密的城市——太子港。basemap还可以叠加来自shapefile的地图数据。我先下载了一个带有太子港道路的shapefile（参见http://cegrp.cga.harvard.edu/haiti/?q=resources_data）。Basemap对象有一个非常方便的readshapefile方法，于是在解压完道路数据文件之后，我只在代码中加以下几行就可以了：

shapefile_path = 'ch08/PortAuPrince_Roads/PortAuPrince_Roads'm.readshapefile(shapefile_path, 'roads')

在对经纬度边界进行了一番尝试之后，我做了一张反映食物短缺情况的图片，如图8-25所示。

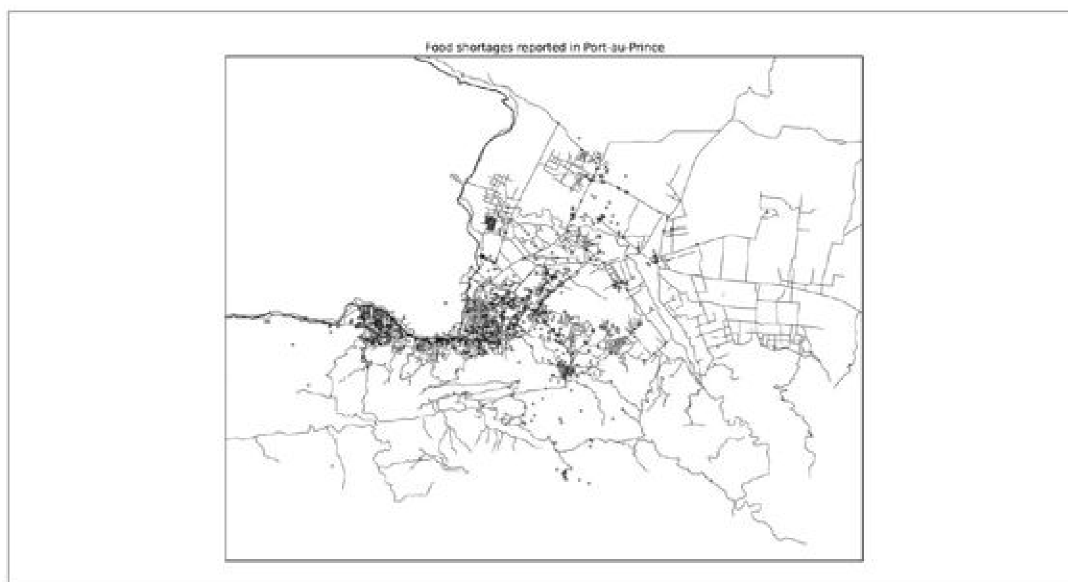


图8-25：海地大地震期间，太子港的食物短缺报告

译注5：读者就当做两个吧。

Python图形化工具生态系统

用Python创建图形的方式非常多（根本罗列不完）。除了开源库，商业库也不少。

本书主要涉及的是matplotlib，因为它是Python领域中使用最广泛的绘图工具。虽然matplotlib是Python科学计算生态系统的重要组成部分，但它在统计图表的创建和展示方面仍然有许多缺点。MATLAB用户可能会对matplotlib感到熟悉，而R用户（尤其是使用ggplot2和trellis的那些）可能就会比较郁闷了（至少目前是）。虽然matplotlib可以为Web应用创建漂亮的图表，但这通常需要耗费大量的精力，因为它原本是为印刷而设计的。先不管美不美观，至少它足以应付大部分需求。在pandas中，我跟其他开发人员一直都在寻求使数据分析中的大部分绘图工作变得更简单的办法。

广泛使用的图形化工具很多。这里我只列举几个，但建议你研究一下整个生态系统。

Chaco

Chaco（http://code.enthought.com/chaco/）是由Enthought开发的一个绘图工具包，它既可以绘制静态图又可以生成交互式图形，如图8-26所示。它非常适合用复杂的图形化方式表达数据的内部关系。跟matplotlib相比，Chaco对交互的支持要好得多，而且渲染速度很快。如果要创建交互式的GUI应用程序，它确实是个不错的选择。

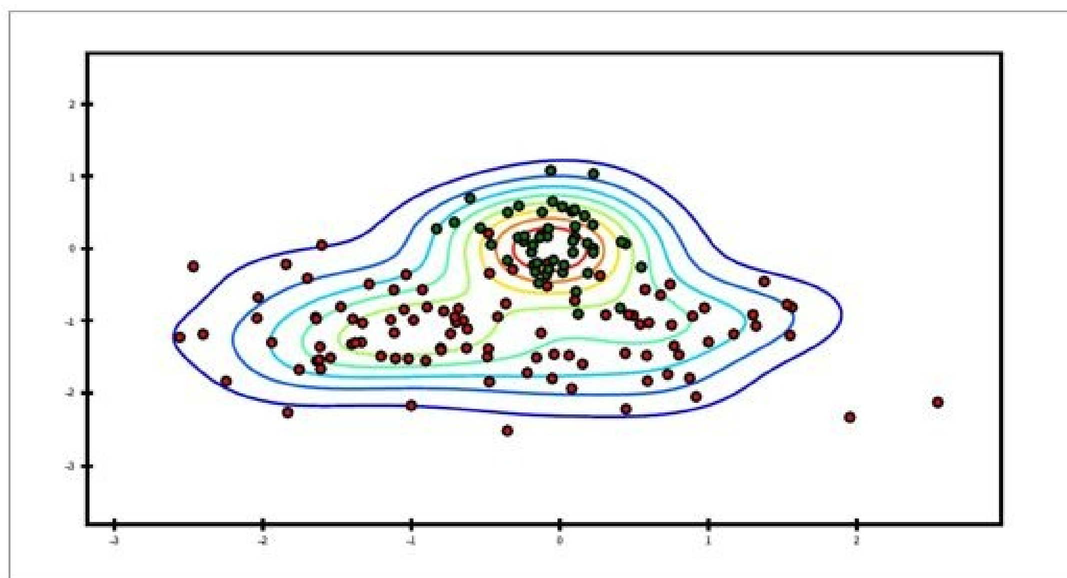


图8-26：Chaco图示例

mayavi

mayavi项目（由Prabhu Ramachandran、Gal Varoquaux等人开发）是一个基于开源C++图形库VTK的3D图形工具包。跟matplotlib一样，mayavi也能集成到IPython以实现交互式使用。通过鼠标和键盘操作，图形可以被平移、旋转、缩放。在第12章中，我用mayavi制作了一张有关广播的插图。我没有给出任何调用mayavi的代码，但你可以在网上找到很多文档和示例。我相信它能成为WebGL（以及相关产品）的替代品，虽然其生成的图形很难以交互的形式共享。

其他库

当然，Python领域中还有许多其他的图形化库和应用程序：PyQwt、Veusz、gnuplot-py、biggles等。我就曾经见过PyQwt被用在基于Qt框架（PyQt）的GUI应用程序中。许多库都还在不断地发展（有些已经被用在大型应用程序当中了）。近几年来，我发现了一个总体趋势：大部分库都在向基于Web的技术发展，并逐渐远离桌面图形技术。下面我要就这个问题多说几句。

图形化工具的未来

基于Web技术（比如JavaScript）的图形化是必然的发展趋势。毫无疑问，许多基于Flash或JavaScript的静态或交互式图形化工具已经出现了很多年。而且类似的新工具包（如d3.js及其分支项目）一直都在不断涌现。相比之下，非Web式的图形化开发工作在近几年中减慢了许多。Python以及其他数据分析和统计计算环境（如R）都是如此。

于是，开发方向就变成了实现数据分析和准备工具（如pandas）与Web浏览器之间更为紧密的集成。我希望这个思路今后能成为Python以及非Python用户之间富有成效的协作手段。

第9章　数据聚合与分组运算

对数据集进行分组并对各组应用一个函数（无论是聚合还是转换），这是数据分析工作中的重要环节。在将数据集准备好之后，通常的任务就是计算分组统计或生成视表。pandas提供了一个灵活高效的gruopby功能，它使你能以一种自然的方式对数据集进行切片、切块、摘要等操作。

关系型数据库和SQL（Structured Query Language，结构化查询语言）能够如此流行的原因之一就是其能够方便地对数据进行连接、过滤、转换和聚合。但是，像SQL这样的查询语言所能执行的分组运算的种类很有限。在本章中你将会看到，由于Python和pandas强大的表达能力，我们可以执行复杂得多的分组运算（利用任何可以接受pandas对象或NumPy数组的函数）。在本章中，你将会学到：

·根据一个或多个键（可以是函数、数组或DataFrame列名）拆分pandas对象。

·计算分组摘要统计，如计数、平均值、标准差，或用户自定义函数。

·对DataFrame的列应用各种各样的函数。

·应用组内转换或其他运算，如规格化、线性回归、排名或选取子集等。

·计算透视表或交叉表。

·执行分位数分析以及其他分组分析。

注意：对时间数据的聚合（groupby的特殊用法之一）也称作重采样（resampling），本书将在第10章中单独对其进行讲解。

GroupBy技术

Hadley Wickham（许多热门R语言包的作者）创造了一个用于表示分组运算的术语"split-apply-combine"（拆分－应用－合并），我觉得这个词很好地描述了整个过程。分组运算的第一个阶段，pandas象（无论是Series、DataFrame还是其他的）中的数据会根据你所提供的一个或多个键被拆分（split）为多组。拆分操作是在对象的特定轴上执行的。例如，DataFrame可以在其行（axis=0）或列（axis=1）上进行分组。然后，将一个函数应用（apply）到各个分组并产生一个新值。最后，所有这些函数的执行结果会被合并（combine）到最终的结果对象中。结果对象的形式一般取决于数据上所执行的操作。图9-1大致说明了一个简单的分组聚合过程。

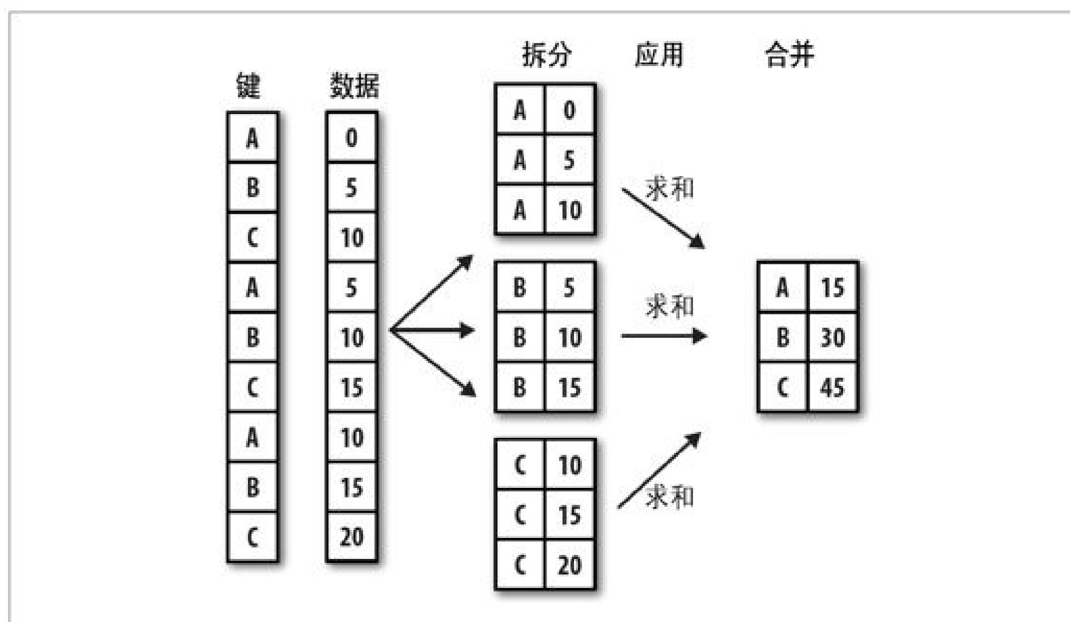


图9-1：分组聚合演示

分组键可以有多种形式，且类型不必相同：

·列表或数组，其长度与待分组的轴一样。

·表示DataFrame某个列名的值。

·字典或Series，给出待分组轴上的值与分组名之间的对应关系。

·函数，用于处理轴索引或索引中的各个标签。

注意，后三种都只是快捷方式而已，其最终目的仍然是产生一组用于拆分对象的值。如果觉得这些东西看起来很抽象，不用担心，我将在本章中给出大量有关于此的示例。首先来看看下面这个非常简单的表格型数据集（以DataFrame的形式）：

In [13]: df = DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'], ...: 'key2' : ['one', 'two', 'one', 'two', 'one'], ...: 'data1' : np.random.randn(5), ...: 'data2' : np.random.randn(5)})In [14]: dfOut[14]: data1 data2 key1 key20 -0.204708 1.393406 a one1 0.478943 0.092908 a two2 -0.519439 0.281746 b one3 -0.555730 0.769023 b two4 1.965781 1.246435 a one

假设你想要按key1进行分组，并计算data1列的平均值。实现该功能的方式有很多，而我们这里要用的是：访问data1，并根据key1调用groupby：

In [15]: grouped = df['data1'].groupby(df['key1'])In [16]: groupedOut[16]:

变量grouped是一个GroupBy对象。它实际上还没有进行任何计算，只是含有一些有关分组键df['key1']的中间数据而已。换句话说，该对象已经有了接下来对各分组执行运算所需的一切信息。例如，我们可以调用GroupBy的mean方法来计算分组平均值：

In [17]: grouped.mean()Out[17]:key1a 0.746672b -0.537585

稍后我将详细讲解.mean()的调用过程。这里最重要的是，数据（Series）根据分组键进行了聚合，产生了一个新的Series，其索引为key1列中的唯一值。之所以结果中索引的名称为key1，是因为原始DataFrame的列df['key1']就叫这个名字。

如果我们一次传入多个数组，就会得到不同的结果：

In [18]: means = df['data1'].groupby([df['key1'], df['key2']]).mean()In [19]: meansOut[19]:key1 key2a one 0.880536 two 0.478943b one -0.519439 two -0.555730

这里，我通过两个键对数据进行了分组，得到的Series具有一个层次化索引（由唯一的键对组成）：

In [20]: means.unstack()Out[20]:key2 one twokey1a 0.880536 0.478943b -0.519439 -0.555730

在上面这些示例中，分组键均为Series。实际上，分组键可以是任何长度适当的数组：

In [21]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])In [22]: years = np.array([2005, 2005, 2006, 2005, 2006])In [23]: df['data1'].groupby([states, years]).mean()Out[23]:California 2005 0.478943 2006 -0.519439Ohio 2005 -0.380219 2006 1.965781

此外，你还可以将列名（可以是字符串、数字或其他Python对象）用作分组键：

In [24]: df.groupby('key1').mean()Out[24]: data1 data2key1a 0.746672 0.910916b -0.537585 0.525384In [25]: df.groupby(['key1', 'key2']).mean()Out[25]: data1 data2key1 key2a one 0.880536 1.319920 two 0.478943 0.092908b one -0.519439 0.281746 two -0.555730 0.769023

你可能已经注意到了，在执行df.groupby('key1').mean()时，结果中没有key2列。这是因为df['key2']不是数值数据（俗称“麻烦列”），所以被从结果中排除了。默认情况下，所有数值列都会被聚合，虽然有时可能会被过滤为一个子集（稍后就会讲到）。

无论你准备拿groupby做什么，都有可能会用到GroupBy的size方法，它可以返回一个含有分组大小的Series：

In [26]: df.groupby(['key1', 'key2']).size()Out[26]:key1 key2a one 2 two 1b one 1 two 1

警告：到编写本书时为止，分组键中的任何缺失值都会被排除在结果之外。在你读到这里的时候，说不定就已经有一个选项可以使结果中包含NA组了

译注1

。

对分组进行迭代

GroupBy对象支持迭代，可以产生一组二元元组（由分组名和数据块组成）。看看下面这个单的数据集：

In [27]: for name, group in df.groupby('key1'): ...: print name ...: print group ...:a data1 data2 key1 key20 -0.204708 1.393406 a one1 0.478943 0.092908 a two4 1.965781 1.246435 a oneb data1 data2 key1 key22 -0.519439 0.281746 b one3 -0.555730 0.769023 b two

对于多重键的情况，元组的第一个元素将会是由键值组成的元组：

In [28]: for (k1, k2), group in df.groupby(['key1', 'key2']): ...: print k1, k2 ...: print group ...:a one data1 data2 key1 key20 -0.204708 1.393406 a one4 1.965781 1.246435 a onea two data1 data2 key1 key21 0.478943 0.092908 a twob one data1 data2 key1 key22 -0.519439 0.281746 b oneb two data1 data2 key1 key23 -0.55573 0.769023 b two

当然，你可以对这些数据片段做任何操作。有一个你可能会觉得有用的运算：将这些数据片段做成一个字典。

In [29]: pieces = dict(list(df.groupby('key1')))In [30]: pieces['b']Out[30]: data1 data2 key1 key22 -0.519439 0.281746 b one3 -0.555730 0.769023 b two

groupby默认是在axis=0上进行分组的，通过设置也可以在其他任何轴上进行分组。拿上面例子中的df来说，我们可以根据dtype对列进行分组：

In [31]: df.dtypesOut[31]:data1 float64data2 float64key1 objectkey2 objectIn [32]: grouped = df.groupby(df.dtypes, axis=1)In [33]: dict(list(grouped))Out[33]:{dtype('float64'): data1 data20 -0.204708 1.3934061 0.478943 0.0929082 -0.519439 0.2817463 -0.555730 0.7690234 1.965781 1.246435, dtype('object'): key1 key20 a one1 a two2 b one3 b two4 a one}

选取一个或一组列

对于由DataFrame产生的GroupBy对象，如果用一个（单个字符串）或一组（字符串数组）列名对其进行索引，就能实现选取部分列进行聚合的目的。也就是说：

df.groupby('key1')['data1']df.groupby('key1')[['data2']]

是以下代码的语法糖：

df['data1'].groupby(df['key1'])df[['data2']].groupby(df['key1'])

尤其对于大数据集，很可能只需要对部分列进行聚合。例如，在前面那个数据集中，如果只需计算data2列的平均值并以DataFrame形式得到结果，我们可以编写：

In [34]: df.groupby(['key1', 'key2'])[['data2']].mean()Out[34]: data2key1 key2a one 1.319920 two 0.092908b one 0.281746 two 0.769023

这种索引操作所返回的对象是一个已分组的DataFrame（如果传入的是列表或数组）或已分组的Series（如果传入的是标量形式的单个列名）：

In [35]: s_grouped = df.groupby(['key1', 'key2'])['data2']In [36]: s_groupedOut[36]: In [37]: s_grouped.mean()Out[37]:key1 key2a one 1.319920 two 0.092908b one 0.281746 two 0.769023Name: data2

通过字典或Series进行分组

除数组以外，分组信息还可以其他形式存在。来看另一个示例DataFrame：

In [38]: people = DataFrame(np.random.randn(5, 5), ...: columns=['a', 'b', 'c', 'd', 'e'], ...: index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])In [39]: people.ix[2:3, ['b', 'c']] = np.nan # 添加几个NA值In [40]: peopleOut[40]: a b c d eJoe 1.007189 -1.296221 0.274992 0.228913 1.352917Steve 0.886429 -2.001637 -0.371843 1.669025 -0.438570Wes -0.539741 NaN NaN -1.021228 -0.577087Jim 0.124121 0.302614 0.523772 0.000940 1.343810Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757

假设已知列的分组关系，并希望根据分组计算列的总计：

In [41]: mapping = {'a': 'red', 'b': 'red', 'c': 'blue', ...: 'd': 'blue', 'e': 'red', 'f' : 'orange'}

现在，只需将这个字典传给groupby即可：

In [42]: by_column = people.groupby(mapping, axis=1)In [43]: by_column.sum()Out[43]: blue redJoe 0.503905 1.063885Steve 1.297183 -1.553778Wes -1.021228 -1.116829Jim 0.524712 1.770545Travis -4.230992 -2.405455

Series也有同样的功能，它可以被看做一个固定大小的映射。对于上面那个例子，如果用Series作为分组键，则pandas会检查Series以确保其索引跟分组轴是对齐的：

In [44]: map_series = Series(mapping)In [45]: map_seriesOut[45]:a redb redc blued bluee redf orangeIn [46]: people.groupby(map_series, axis=1).count()Out[46]: blue redJoe 2 3Steve 2 3Wes 1 2Jim 2 3Travis 2 3

通过函数进行分组

相较于字典或Series，Python函数在定义分组映射关系时可以更有创意且更为抽象。任何被当做分组键的函数都会在各个索引值上被调用一次，其返回值就会被用作分组名称。具体点说，以上一小节的示例DataFrame为例，其索引值为人的名字。假设你希望根据人名的长度进行分组，虽然可以求取一个字符串长度数组，但其实仅仅传入len函数就可以了：

In [47]: people.groupby(len).sum()Out[47]: a b c d e3 0.591569 -0.993608 0.798764 -0.791374 2.1196395 0.886429 -2.001637 -0.371843 1.669025 -0.4385706 -0.713544 -0.831154 -2.370232 -1.860761 -0.860757

将函数跟数组、列表、字典、Series混合使用也不是问题，因为任何东西最终都会被转换为数组：

In [48]: key_list = ['one', 'one', 'one', 'two', 'two']In [49]: people.groupby([len, key_list]).min()Out[49]: a b c d e3 one -0.539741 -1.296221 0.274992 -1.021228 -0.577087 two 0.124121 0.302614 0.523772 0.000940 1.3438105 one 0.886429 -2.001637 -0.371843 1.669025 -0.4385706 two -0.713544 -0.831154 -2.370232 -1.860761 -0.860757

根据索引级别分组

层次化索引数据集最方便的地方就在于它能够根据索引级别进行聚合。要实现该目的，通过level关键字传入级别编号或名称即可：

In [50]: columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'], ...: [1, 3, 5, 1, 3]], names=['cty', 'tenor'])In [51]: hier_df = DataFrame(np.random.randn(4, 5), columns=columns)In [52]: hier_dfOut[52]:cty US JPtenor 1 3 5 1 30 0.560145 -1.265934 0.119827 -1.063512 0.3328831 -2.359419 -0.199543 -1.541996 -0.970736 -1.3070302 0.286350 0.377984 -0.753887 0.331286 1.3497423 0.069877 0.246674 -0.011862 1.004812 1.327195In [53]: hier_df.groupby(level='cty', axis=1).count()Out[53]:cty JP US0 2 31 2 32 2 33 2 3

译注1

：翻译本书过程中仍然没有。

数据聚合

对于聚合，我指的是任何能够从数组产生标量值的数据转换过程。之前的例子中我已经用过一些，比如mean、count、min以及sum等。你可能想知道在GroupBy对象上调用mean()时究竟发生了什么。多常见的聚合运算（如表9-1所示）都有就地计算数据集统计信息的优化实现。然而，并不是只能使用这些方法。你可以使用自己发明的聚合运算，还可以调用分组对象上已经定义好的任何方法。例如，quantile可以计算Series或DataFrame列的样本分位数

译注2

：

In [54]: dfOut[54]: data1 data2 key1 key20 -0.204708 1.393406 a one1 0.478943 0.092908 a two2 -0.519439 0.281746 b one3 -0.555730 0.769023 b two4 1.965781 1.246435 a oneIn [55]: grouped = df.groupby('key1')In [56]: grouped['data1'].quantile(0.9)Out[56]:key1a 1.668413b -0.523068

虽然quantile并没有明确地实现于GroupBy，但它是一个Series方法，所以这里是能用的。实际上，GroupBy会高效地对Series进行切片，然后对各片调用piece.quantile(0.9)，最后将这些结果组装成最终结果。

如果要使用你自己的聚合函数，只需将其传入aggregate或agg方法即可：

In [57]: def peak_to_peak(arr): ...: return arr.max() - arr.min()In [58]: grouped.agg(peak_to_peak)Out[58]: data1 data2key1a 2.170488 1.300498b 0.036292 0.487276

注意，有些方法（如describe）也是可以用在这里的，即使严格来讲，它们并非聚合运算：

In [59]: grouped.describe()Out[59]: data1 data2key1a count 3.000000 3.000000 mean 0.746672 0.910916 std 1.109736 0.712217 min -0.204708 0.092908 25% 0.137118 0.669671 50% 0.478943 1.246435 75% 1.222362 1.319920 max 1.965781 1.393406b count 2.000000 2.000000 mean -0.537585 0.525384 std 0.025662 0.344556 min -0.555730 0.281746 25% -0.546657 0.403565 50% -0.537585 0.525384 75% -0.528512 0.647203 max -0.519439 0.769023

在后面关于分组级运算

译注3

和转换的那一节中，我将详细说明这到底是怎么回事。

注意： 可能你已经注意到了，自定义聚合函数要比表9-1中那些经过优化的函数慢得多这是因为在构造中间分组数据块时存在非常大的开销（函数调用、数据重排等）。

表9-1：经过优化的groupby[译注4]的方法

函数名	说明
count	分组中非NA值的数量
sum	非NA值的和
mean	非NA值的平均值
median	非NA值的算术中位数
std、var	无偏（分母为n - 1）标准差和方差
min、max	非NA值的最小值和最大值
prod	非NA值的积
first、last	第一个和最后一个非NA值

译注4：这里应该是“经过优化的GroupBy的方法”，原文有误。

为了说明一些更高级的聚合功能，我将使用一个有关餐馆小费的数据集。我是在R语言的reshape2包中得到该数据集的（可以在本书的GitHub库中找到）。它最初出现于Bryant和Smith在1995年编写的一本有关商业统计的书中。通过read_csv将其加载之后，我添加了一个表示小费比例的列tip_pct。

In [60]: tips = pd.read_csv('ch08/tips.csv')# 添加“小费占总额百分比”的列In [61]: tips['tip_pct'] = tips['tip'] / tips['total_bill']In [62]: tips[:6]Out[62]: total_bill tip sex smoker day time size tip_pct0 16.99 1.01 Female False Sun Dinner 2 0.0594471 10.34 1.66 Male False Sun

Dinner 3 0.1605422 21.01 3.50 Male False Sun Dinner 3 0.1665873 23.68 3.31 Male False Sun Dinner 2 0.1397804 24.59 3.61 Female False Sun Dinner 4 0.1468085 25.29 4.71 Male False Sun Dinner 4 0.186240

面向列的多函数应用

我们已经看到，对Series或DataFrame列的聚合运算其实就是使用aggregate（使用自定义函数）或调用诸如mean、std之类的方法。然而，你可能希望对不同的列使用不同的聚合函数，或一次应用多个函数。其实这事也好办，我将通过一些示例来进行讲解。首先，我根据sex和smoker对tips进行分组：

In [63]: grouped = tips.groupby(['sex', 'smoker'])

注意，对于表9-1中的那些描述统计，可以将函数名以字符串的形式传入：

In [64]: grouped_pct = grouped['tip_pct']In [65]: grouped_pct.agg('mean')Out[65]:sex smokerFemale False 0.156921 True 0.182150Male False 0.160669 True 0.152771Name: tip_pct

如果传入一组函数或函数名，得到的DataFrame的列就会以相应的函数命名：

In [66]: grouped_pct.agg(['mean', 'std', peak_to_peak])Out[66]: mean std peak_to_peaksex smokerFemale False 0.156921 0.036421 0.195876 True 0.182150 0.071595 0.360233Male False 0.160669 0.041849 0.220186 True 0.152771 0.090588 0.674707

你并非一定要接受GroupBy自动给出的那些列名，特别是lambda函数，它们的名称是'<lambda>'，这样的辨识度就很低了（通过函数的name属性看看就知道了）。如果传入的是一个由(name,function)元组组成的列表，则各元组的第一个元素就会被用作DataFrame的列名（可以将这种二元元组列表看做一个有序映射）：

In [67]: grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])Out[67]: foo barsex smokerFemale False 0.156921 0.036421 True 0.182150 0.071595Male False 0.160669 0.041849 True 0.152771 0.090588

对于DataFrame，你还可以定义一组应用于全部列的函数，或不同的列应用不同的函数。假设我们想要对tip_pct和total_bill列计算三个统计信息：

In [68]: functions = ['count', 'mean', 'max']In [69]: result = grouped['tip_pct', 'total_bill'].agg(functions)In [70]: resultOut[70]: tip_pct total_bill count mean max count mean maxsex smokerFemale False 54 0.156921 0.252672 54 18.105185 35.83 True 33 0.182150 0.416667 33 17.977879 44.30Male False 97 0.160669 0.291990 97 19.791237 48.33 True 60 0.152771 0.710345 60 22.284500 50.81

如你所见，结果DataFrame拥有层次化的列，这相当于分别对各列进行聚合，然后用concat将结果组装到一起（列名用作keys参数）。

In [71]: result['tip_pct']Out[71]: count mean maxsex smokerFemale False 54 0.156921 0.252672 True 33 0.182150 0.416667Male False 97 0.160669 0.291990 True 60 0.152771 0.710345

跟前面一样，这里也可以传入带有自定义名称的元组列表：

In [72]: ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]In [73]: grouped['tip_pct', 'total_bill'].agg(ftuples)Out[73]: tip_pct total_bill Durchschnitt Abweichung Durchschnitt Abweichungsex smokerFemale False 0.156921 0.001327 18.105185 53.092422 True 0.182150 0.005126 17.977879 84.451517Male False 0.160669 0.001751 19.791237 76.152961 True 0.152771 0.008206 22.284500 98.244673

现在，假设你想要对不同的列应用不同的函数。具体的办法是向agg传入一个从列名映射到函数的字典：

In [74]: grouped.agg({'tip' : np.max, 'size' : 'sum'})Out[74]: size tipsex smokerFemale False 140 5.2 True 74 6.5Male False 263 9.0 True 150 10.0In [75]: grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'], ...: 'size' : 'sum'})Out[75]: tip_pct size min max mean std sumsex smokerFemale False 0.056797 0.252672 0.156921 0.036421 140 True 0.056433 0.416667 0.182150 0.071595 74Male False 0.071804 0.291990 0.160669 0.041849 263 True 0.035638 0.710345 0.152771 0.090588 150

只有将多个函数应用到至少一列时，DataFrame才会拥有层次化的列。

以“无索引”的形式返回聚合数据

到目前为止，所有示例中的聚合数据都有由唯一的分组键组成的索引（可能还是层次化的）。由于并不总是需要如此，所以你可以向groupby传入as_index=False以禁用该功能：

In [76]: tips.groupby(['sex', 'smoker'], as_index=False).mean()Out[76]: sex smoker total_bill tip size tip_pct0 Female False 18.105185 2.773519 2.592593 0.1569211 Female True 17.977879 2.931515 2.242424 0.1821502 Male False 19.791237 3.113402 2.711340 0.1606693 Male True 22.284500 3.051167 2.500000 0.152771

当然，对结果调用reset_index也能得到这种形式的结果。

警告： groupby的这种用法比较缺乏灵活性。

译注2：注意，如果传入的百分位上没有值，则quantile会进行线性插值。

译注3

：也就是“面向分组的计算。

分组级运算和转换

聚合只不过是分组运算的其中一种而已。它是数据转换的一个特例，也就是说，它接受能够将一维数组简化为标量值的函数。在本节中，我将介绍transform和apply方法，它们能够执行更多其他的分组运算。

假设我们想要为一个DataFrame添加一个用于存放各索引分组平均值的列。一个办法是先聚合再合并：

In [77]: dfOut[77]: data1 data2 key1 key20 -0.204708 1.393406 a one1 0.478943 0.092908 a two2 -0.519439 0.281746 b one3 -0.555730 0.769023 b two4 1.965781 1.246435 a oneIn [78]: k1*means = df.groupby('key1').mean().add_prefix('mean*')In [79]: k1_meansOut[79]: mean_data1 mean_data2key1a 0.746672 0.910916b -0.537585 0.525384In [80]: pd.merge(df, k1_means, left_on='key1', right_index=True)Out[80]: data1 data2 key1 key2 mean_data1 mean_data20 -0.204708 1.393406 a one 0.746672 0.9109161 0.478943 0.092908 a two 0.746672 0.9109164 1.965781 1.246435 a one 0.746672 0.9109162 -0.519439 0.281746 b one -0.537585 0.5253843 -0.555730 0.769023 b two -0.537585 0.525384

虽然这样也行，但是不太灵活。你可以将该过程看做利用np.mean函数对两个数据列进行转换。再以本章前面用过的那个people DataFrame为例，这次我们在GroupBy上使用transform方法：

In [81]: key = ['one', 'two', 'one', 'two', 'one']In [82]: people.groupby(key).mean()Out[82]: a b c d eone -0.082032 -1.063687 -1.047620 -0.884358 -0.028309two 0.505275 -0.849512 0.075965 0.834983 0.452620In [83]: people.groupby(key).transform(np.mean)Out[83]: a b c d eJoe -0.082032 -1.063687 -1.047620 -0.884358 -0.028309Steve 0.505275 -0.849512 0.075965 0.834983 0.452620Wes -0.082032 -1.063687 -1.047620 -0.884358 -0.028309Jim 0.505275 -0.849512 0.075965 0.834983 0.452620Travis -0.082032 -1.063687 -1.047620 -0.884358 -0.028309

不难看出，transform会将一个函数应用到各个分组，然后将结果放置到适当的位置上。如果各分组产生的是一个标量值，则该值就会被广播出去。现在，假设你希望从各组中减去平均值。为此，我们先创建一个距平化函数（demeaning function），然后将其传给transform：

In [84]: def demean(arr): ...: return arr - arr.mean()In [85]: demeaned = people.groupby(key).transform(demean)In [86]: demeanedOut[86]: a b c d eJoe 1.089221 -0.232534 1.322612 1.113271 1.381226Steve 0.381154 -1.152125 -0.447807 0.834043 -0.891190Wes -0.457709 NaN NaN -0.136869 -0.548778Jim -0.381154 1.152125 0.447807 -0.834043 0.891190Travis -0.631512 0.232534 -1.322612 -0.976402 -0.832448

你可以检查一下demeaned现在的分组平均值是否为0：

In [87]: demeaned.groupby(key).mean()Out[87]: a b c d eone 0 -0 0 0 0two -0 0 0 0 0

在下一节中你将会看到，分组距平化操作还可以通过apply实现。

apply：一般性的“拆分－应用－合并”

跟aggregate一样，transform也是一个有着严格条件的特殊函数：传入的函数只能产生两种结果，要么产生一个可以广播的标量值（如np.mean），要么产生一个相同大小的结果数组。最一般化的GroupBy方法是apply，本节剩余部分将重点讲解它。如图9-1所示，apply会将待处理的对象拆分成多个片段，然后对各片段调用传入的函数，最后尝试将各片段组合到一起。

回到之前那个小费数据集，假设你想要根据分组选出最高的5个tip*pct*值。首先，编写一个选取指定列具有最大值的行的函数*[译注5](#809468440711498-BuChongShuo_Ming_Yi_Xia* ***Chang_Du_Xiang_Deng_De_Tong****Zhi_De_Shi****QuJianDa_Xiao_Xiang_Deng****Da_Xiao_Xiang_Deng_De_Tong****Zhi_De_Shi****Shu_Ju_Dian_Shu_Liang_Xiang_Deng*)：

In [88]: def top(df, n=5, column='tip_pct'): ...: return df.sort_index(by=column)[-n:]In [89]: top(tips, n=6)Out[89]: total_bill tip sex smoker day time size tip_pct109 14.31 4.00 Female True Sat Dinner 2 0.279525183 23.17 6.50 Male True Sun Dinner 4 0.280535232 11.61 3.39 Male False Sat Dinner 2 0.29199067 3.07 1.00 Female True Sat Dinner 1 0.325733178 9.60 4.00 Female True Sun Dinner 2 0.416667172 7.25 5.15 Male True Sun Dinner 2 0.710345

现在，如果对smoker分组并用该函数调用apply，就会得到：

In [90]: tips.groupby('smoker').apply(top)Out[90]: total_bill tip sex smoker day time size tip_pctsmokerNo 88 24.71 5.85 Male False Thur Lunch 2 0.236746 185 20.69 5.00 Male False Sun Dinner 5 0.241663 51 10.29 2.60 Female False Sun Dinner 2 0.252672 149 7.51 2.00 Male False Thur Lunch 2 0.266312 232 11.61 3.39 Male False Sat Dinner 2 0.291990Yes 109 14.31 4.00 Female True Sat Dinner 2 0.279525 183 23.17 6.50 Male True Sun Dinner 4 0.280535 67 3.07 1.00 Female True Sat Dinner 1 0.325733 178 9.60 4.00 Female True Sun Dinner 2 0.416667 172 7.25 5.15 Male True Sun Dinner 2 0.710345

这里发生了什么？top函数在DataFrame的各个片段上调用，然后结果由pandas.concat组装到一起，并以分组名称进行了标记。于是，最终结果就有了一个层次化索引，其内层索引值来自原DataFrame。

如果传给apply的函数能够接受其他参数或关键字，则可以将这些内容放在函数名后面一并传入：

In [91]: tips.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')Out[91]: total_bill tip sex smoker day time size tip_pctsmoker dayNo Fri 94 22.75 3.25 Female False Fri Dinner 2 0.142857 Sat 212 48.33 9.00 Male False Sat Dinner 4 0.186220 Sun 156 48.17 5.00 Male False Sun Dinner 6 0.103799 Thur 142 41.19 5.00 Male False Thur Lunch 5 0.121389Yes Fri 95 40.17 4.73 Male True Fri Dinner 4 0.117750 Sat 170 50.81 10.00 Male True Sat Dinner 3 0.196812 Sun 182 45.35 3.50 Male True Sun Dinner 3 0.077178 Thur 197 43.11 5.00 Female True Thur Lunch 4 0.115982

注意： 除这些基本用法之外，能否充分发挥apply的威力很大程度上取决于你的创造力。传入的那个函数能做什么全由你说了算，它只需返回一个pandas对象或标量值即可。本章后续部分的示例主要用于讲解如何利用groupby解决各种各样的问题。

可能你已经想起来了，之前我在GroupBy对象上调用过describe：

In [92]: result = tips.groupby('smoker')['tip_pct'].describe()In [93]: resultOut[93]:smokerNo count 151.000000 mean 0.159328 std 0.039910 min 0.056797 25% 0.136906 50% 0.155625 75% 0.185014 max 0.291990Yes count 93.000000 mean 0.163196 std 0.085119 min 0.035638 25% 0.106771 50% 0.153846 75% 0.195059 max 0.710345In [94]: result.unstack('smoker')Out[94]:smoker No Yescount 151.000000 93.000000mean 0.159328 0.163196std 0.039910 0.085119min 0.056797 0.03563825% 0.136906 0.10677150% 0.155625 0.15384675% 0.185014 0.195059max 0.291990 0.710345

在GroupBy中，当你调用诸如describe之类的方法时，实际上只是应用了下面两条代码的快捷方式而已：

f = lambda x: x.describe()grouped.apply(f)

禁止分组键

从上面的例子中可以看出，分组键会跟原始对象的索引共同构成结果对象中的层次化索引。将group_keys=False传入groupby即可禁止该效果：

In [95]: tips.groupby('smoker', group_keys=False).apply(top)Out[95]: total_bill tip sex smoker day time size tip_pct88 24.71 5.85 Male False Thur Lunch 2 0.236746185 20.69 5.00 Male False Sun Dinner 5 0.24166351 10.29 2.60 Female False Sun Dinner 2 0.252672149 7.51 2.00 Male False Thur Lunch 2 0.266312232 11.61 3.39 Male False Sat Dinner 2 0.291990109 14.31 4.00 Female True Sat Dinner 2 0.279525183 23.17 6.50 Male True Sun Dinner 4 0.28053567 3.07 1.00 Female True Sat Dinner 1 0.325733178 9.60 4.00 Female True Sun Dinner 2 0.416667172 7.25 5.15 Male True Sun Dinner 2 0.710345

分位数和桶分析

我曾在第7章中讲过，pandas有一些能根据指定面元或样本分位数将数据拆分成多块的工具（比如cut和qcut）。将这些函数跟groupby结合起来，就能非常轻松地实现对数据集的桶（bucket）或分位数（quantile）分析了。以下面这个简单的随机数据集为例，我们利用cut将其装入长度相等的桶中：

In [96]: frame = DataFrame({'data1': np.random.randn(1000), ...: 'data2': np.random.randn(1000)})In [97]: factor = pd.cut(frame.data1, 4)In [98]: factor[:10]Out[98]:Categorical:array([(-1.23, 0.489], (-2.956, -1.23], (-1.23, 0.489], (0.489, 2.208], (-1.23, 0.489], (0.489, 2.208], (-1.23, 0.489], (-1.23, 0.489], (0.489, 2.208], (0.489, 2.208]], dtype=object)Levels (4): Index([(-2.956, -1.23], (-1.23, 0.489], (0.489, 2.208], (2.208, 3.928]], dtype=object)

由cut返回的Factor对象可直接用于groupby。因此，我们可以像下面这样对data2做一些统计计算：

In [99]: def get_stats(group): ...: return {'min': group.min(), 'max': group.max(), ...: 'count': group.count(), 'mean': group.mean()}In [100]: grouped = frame.data2.groupby(factor)In [101]: grouped.apply(get_stats).unstack()Out[101]: count max mean mindata1(-1.23, 0.489] 598 3.260383 -0.002051 -2.989741(-2.956, -1.23] 95 1.670835 -0.039521 -3.399312(0.489, 2.208] 297 2.954439 0.081822 -3.745356(2.208, 3.928] 10 1.765640 0.024750 -1.929776

这些都是长度相等的桶。要根据样本分位数得到大小相等的桶，使用qcut即可

译注6

。传入labels=False即可只获取分位数的编号。

返回分位数编号In [102]: grouping = pd.qcut(frame.data1, 10, labels=False)In [103]: grouped = frame.data2.groupby(grouping)In [104]: grouped.apply(get_stats).unstack()Out[104]: count max mean min0 100 1.670835 -0.049902 -3.3993121 100 2.628441 0.030989 -1.9500982 100 2.527939 -0.067179 -2.9251133 100 3.260383 0.065713 -2.3155554 100 2.074345 -0.111653 -2.0479395 100 2.184810 0.052130 -2.9897416 100 2.458842 -0.021489 -2.2235067 100 2.954439 -0.026459 -3.0569908 100 2.735527 0.103406 -3.7453569 100 2.377020 0.220122 -2.064111

示例：用特定于分组的值填充缺失值

对于缺失数据的清理工作，有时你会用dropna将其滤除，而有时则可能会希望用一个固定值或由数据集本身所衍生出来的值去填充NA值。这时就得使用fillna这个工具了。在下面这个例子中，我用平均值去填充NA值：

In [105]: s = Series(np.random.randn(6))In [106]: s[::2] = np.nanIn [107]: sOut[107]:0 NaN1 -0.1259212 NaN3 -0.8844754 NaN5 0.227290In [108]: s.fillna(s.mean())Out[108]:0 -0.2610351 -0.1259212 -0.2610353 -0.8844754 -0.2610355 0.227290

假设你需要对不同的分组填充不同的值。可能你已经猜到了，只需将数据分组，并使用apply和一个能够对各数据块调用fillna的函数即可。下面是一些有关美国几个州的示例数据，这些州又被分为东部和西部：

In [109]: states = ['Ohio', 'New York', 'Vermont', 'Florida', ...: 'Oregon', 'Nevada', 'California', 'Idaho']In [110]: group_key = ['East'] *4 + ['West']* 4In [111]: data = Series(np.random.randn(8), index=states)In [112]: data[['Vermont', 'Nevada', 'Idaho']] = np.nanIn [113]: dataOut[113]:Ohio 0.922264New York -2.153545Vermont NaNFlorida -0.375842Oregon 0.329939Nevada NaNCalifornia 1.105913Idaho NaNIn [114]: data.groupby(group_key).mean()Out[114]:East -0.535707West 0.717926

我们可以用分组平均值去填充NA值，如下所示：

In [115]: fill_mean = lambda g: g.fillna(g.mean())In [116]: data.groupby(group_key).apply(fill_mean)Out[116]:Ohio 0.922264New York -2.153545Vermont -0.535707Florida -0.375842Oregon 0.329939Nevada 0.717926California 1.105913Idaho 0.717926

此外，也可以在代码中预定义各组的填充值。由于分组具有一个name属性，所以我们可以拿来用一下：

In [117]: fill_values = {'East': 0.5, 'West': -1}In [118]: fill_func = lambda g: g.fillna(fill_values[g.name])In [119]: data.groupby(group_key).apply(fill_func)Out[119]:Ohio 0.922264New York -2.153545Vermont 0.500000Florida -0.375842Oregon 0.329939Nevada -1.000000California 1.105913Idaho -1.000000

示例：随机采样和排列

假设你想要从一个大数据集中随机抽取样本以进行蒙特卡罗模拟（Monte Carlo simulation）或其他分析工作。“抽取”的方式有很多，其中一些的效率会比其他的高很多。一个办法是，选取np.random.permutation(N)的前K个元素，其中N为完整数据的大小，K为期望的样本大小。作为一个更有趣的例子，下面是构造一副英语型扑克牌的一个方式：

红桃（Hearts）、黑桃（Spades）、梅花（Clubs）、方片（Diamonds）suits = ['H',

'S', 'C', 'D']card_val = (range(1, 11) + [10] *3) 4*base_names = ['A'] + range(2, 11) + ['J', 'K', 'Q']cards = []for suit in ['H', 'S', 'C', 'D']: cards.extend(str(num) + suit for num in base_names)deck = Series(card_val, index=cards)

现在我有了一个长度为52的Series，其索引为牌名，值则是21点或其他游戏中用于计分的点数（为了简单起见，我当A的点数为1）：

In [121]: deck[:13]Out[121]:AH 12H 23H 34H 45H 56H 67H 78H 89H 910H 10JH 10KH 10QH 10

现在，根据我上面所讲的，从整副牌中抽出5张，代码如下：

In [122]: def draw(deck, n=5): ...: return deck.take(np.random.permutation(len(deck))[:n])In [123]: draw(deck)Out[123]:AD 18C 85H 5KC 102C 2

假设你想要从每种花色中随机抽取两张牌。由于花色是牌名的最后一个字符，所以我们可以据此进行分组，并使用apply：

In [124]: get_suit = lambda card: card[-1] # 只要最后一个字母就可以了In [125]: deck.groupby(get_suit).apply(draw, n=2)Out[125]:C 2C 2 3C 3D KD 10 8D 8H KH 10 3H 3S 2S 2 4S 4# 另一种办法In [126]: deck.groupby(get_suit, group_keys=False).apply(draw, n=2)Out[126]:KC 10JC 10AD 15D 55H 56H 67S 7KS 10

示例：分组加权平均数和相关系数

根据groupby的“拆分－应用－合并”范式，DataFrame的列与列之间或两个Series之间的运算（比如分组加权平均）成为一种标准作业。以下面这个数据集为例，它含有分组键、值以及一些权重值：

In [127]: df = DataFrame({'category': ['a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'], ...: 'data': np.random.randn(8), ...: 'weights': np.random.rand(8)})In [128]: dfOut[128]: category data weights0 a 1.561587 0.9575151 a 1.219984 0.3472672 a -0.482239 0.5813623 a 0.315667 0.2170914 b -0.047852 0.8944065 b -0.454145 0.9185646 b -0.556774 0.2778257 b 0.253321 0.955905

然后可以利用category计算分组加权平均数：

In [129]: grouped = df.groupby('category')In [130]: get_wavg = lambda g: np.average(g['data'], weights=g['weights'])In [131]: grouped.apply(get_wavg)Out[131]:categorya 0.811643b -0.122262

这个例子比较无聊，所以再看一个稍微实际点的例子——来自Yahoo!Finance的数据集，其中含有标准普尔500指数（SPX字段）和几只股票的收盘价：

In [132]: close_px = pd.read_csv('ch09/stock_px.csv', parse_dates=True, index_col=0)In [133]: close_pxOut[133]:DatetimeIndex: 2214 entries, 2003-01-02 00:00:00 to 2011-10-14 00:00:00Data columns:AAPL 2214 non-null valuesMSFT 2214 non-null valuesXOM 2214 non-null valuesSPX 2214 non-null valuesdtypes: float64(4)In [134]: close_px[-4:]Out[134]: AAPL MSFT XOM SPX2011-10-11 400.29 27.00 76.27 1195.542011-10-12 402.19 26.96 77.16 1207.252011-10-13 408.43 27.18 76.37 1203.662011-10-14 422.00 27.27 78.11 1224.58

来做一个比较有趣的任务：计算一个由日收益率（通过百分数变化计算）与SPX之间的年度相关系数组成的DataFrame。下面是一个实现办法：

In [135]: rets = close_px.pct_change().dropna()In [136]: spx_corr = lambda x: x.corrwith(x['SPX'])In [137]: by_year = rets.groupby(lambda x: x.year)In [138]: by_year.apply(spx_corr)Out[138]: AAPL MSFT XOM SPX2003 0.541124 0.745174 0.661265 12004 0.374283 0.588531 0.557742 12005 0.467540 0.562374 0.631010 12006 0.428267 0.406126 0.518514 12007 0.508118 0.658770 0.786264 12008 0.681434 0.804626 0.828303 12009 0.707103 0.654902 0.797921 12010 0.710105 0.730118 0.839057 12011 0.691931 0.800996 0.859975 1

当然，你还可以计算列与列之间的相关系数：

苹果和微软的年度相关系数In [139]: by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))Out[139]:2003 0.4808682004 0.2590242005 0.3000932006 0.1617352007 0.4177382008 0.6119012009 0.4327382010 0.5719462011 0.581987

示例：面向分组的线性回归

顺着上一个例子继续，你可以用groupby执行更为复杂的分组统计分析，只要函数返回的是pandas对象或标量值即可。例如，我可以定义下面这个regress函数（利用statsmodels库）对各数据块执行普通最小二乘法（Ordinary Least Squares，OLS）回归：

import statsmodels.api as smdef regress(data, yvar, xvars): Y = data[yvar] X = data[xvars] X['intercept'] = 1. result = sm.OLS(Y, X).fit() return result.params

现在，为了按年计算AAPL对SPX收益率的线性回归，我执行：

In [141]: by_year.apply(regress, 'AAPL', ['SPX'])Out[141]: SPX intercept2003 1.195406 0.0007102004 1.363463 0.0042012005 1.766415 0.0032462006 1.645496 0.0000802007 1.198761 0.0034382008 0.968016 -0.0011102009 0.879103 0.0029542010 1.052608 0.0012612011 0.806605 0.001514

译注5

：原文比较拗口，其实就是“在指定列找出最大值，然后把这个值所在的行选取出来”。

译注6

：补充说明一下。“长度相等的桶”指的是“区间大小相等”，“大小相等的桶”指的是“数据点数量相等”。

透视表和交表

透视表（pivot table）是各种电子表格程序和其他数据分析软件中一种常见的数据汇总工具。它根据一个或多个键对数据进行聚合，并根据行和列上的分组键将数据分配到各个矩形区域中。在Python和pandas中，可以通过本章所介绍的groupby功能以及（能够利用层次化索引的）重塑运算制作透视表。DataFrame有一个pivot_table方法，此外还有一个顶级的pandas.pivot_table函数。除能为groupby提供便利之外，pivot_table还可以添加分项小计（也叫做margins）。

回到小费数据集，假设我想要根据sex和smoker计算分组平均数（pivot_table的默认聚合类型），并将sex和smoker放到行上：

In [142]: tips.pivot_table(rows=['sex', 'smoker'])Out[142]: size tip tip_pct total_billsex smokerFemale No 2.592593 2.773519 0.156921 18.105185 Yes 2.242424 2.931515 0.182150 17.977879Male No 2.711340 3.113402 0.160669 19.791237 Yes 2.500000 3.051167 0.152771 22.284500

这对groupby来说也是很简单的事情。现在，假设我们只想聚合tip_pct和size，而且想根据day进行分组。我将smoker放到列上，把day放到行上：

In [143]: tips.pivot_table(['tip_pct', 'size'], rows=['sex', 'day'], ...: cols='smoker')Out[143]: tip_pct sizesmoker No Yes No Yessex dayFemale Fri 0.165296 0.209129 2.500000 2.000000 Sat 0.147993 0.163817 2.307692 2.200000 Sun 0.165710 0.237075 3.071429

2.500000 Thur 0.155971 0.163073 2.480000 2.428571Male Fri 0.138005 0.144730 2.000000 2.125000 Sat 0.162132 0.139067 2.656250 2.629630 Sun 0.158291 0.173964 2.883721 2.600000 Thur 0.165706 0.164417 2.500000 2.300000

还可以对这个表作进一步的处理，传入margins=True添加分项小计。这将会添加标签为All的行和列，其值对应于单个等级中所有数据的分组统计。在下面这个例子中，All值为平均数：不单独考虑烟民与非烟民（All列），不单独考虑行分组两个级别中的任何单项（All行）。

In [144]: tips.pivot_table(['tip_pct', 'size'], rows=['sex', 'day'], ...: cols='smoker', margins=True)Out[144]: size tip_pctsmoker No Yes All No Yes Allsex dayFemale Fri 2.500000 2.000000 2.111111 0.165296 0.209129 0.199388 Sat 2.307692 2.200000 2.250000 0.147993 0.163817 0.156470 Sun 3.071429 2.500000 2.944444 0.165710 0.237075 0.181569 Thur 2.480000 2.428571 2.468750 0.155971 0.163073 0.157525Male Fri 2.000000 2.125000 2.100000 0.138005 0.144730 0.143385 Sat 2.656250 2.629630 2.644068 0.162132 0.139067 0.151577 Sun 2.883721 2.600000 2.810345 0.158291 0.173964 0.162344 Thur 2.500000 2.300000 2.433333 0.165706 0.164417 0.165276All 2.668874 2.408602 2.569672 0.159328 0.163196 0.160803

要使用其他的聚合函数，将其传给aggfunc即可。例如，使用count或len可以得到有关分组大小的交叉表：

In [145]: tips.pivot_table('tip_pct', rows=['sex', 'smoker'], cols='day', ...: aggfunc=len, margins=True)Out[145]:day Fri Sat Sun Thur Allsex smokerFemale No 2 13 14 25 54 Yes 7 15 4 7 33Male No 2 32 43 20 97 Yes 8 27 15 10 60All 19 87 76 62 244

如果存在空的组合（也就是NA），你可能会希望设置一个fill_value：

In [146]: tips.pivot_table('size', rows=['time', 'sex', 'smoker'], ...: cols='day', aggfunc='sum', fill_value=0)Out[146]:day Fri Sat Sun Thurtime sex smokerDinner Female No 2 30 43 2 Yes 8 33 10 0 Male No 4 85 124 0 Yes 12 71 39 0Lunch Female No 3 0 0 60 Yes 6 0 0 17 Male No 0 0 0 50 Yes 5 0 0 23

pivot_table的参数说明请参见表9-2。

表9-2：pivot_table的参数

参数名	说明
values	待聚合的列的名称。默认聚合所有数值列
rows	用于分组的列名或其他分组键，出现在结果透视表的行
cols	用于分组的列名或其他分组键，出现在结果透视表的列
aggfunc	聚合函数或函数列表，默认为'mean'。可以是任何对groupby有效的函数
fill_value	用于替换结果表中的缺失值
margins	添加行/列小计和总计，默认为False

交叉表：crosstab

交叉表（cross-tabulation，简称crosstab）是一种用于计算分组频率的特殊透视表。下面这个范例数据很典型，取自交叉表的Wikipedia页：

In [150]: dataOut[150]: Sample Gender Handedness0 1 Female Right-handed1 2 Male Left-handed2 3 Female Right-handed3 4 Male Right-handed4 5 Male Left-handed5 6 Male Right-handed6 7 Female Right-handed7 8 Female Left-handed8 9 Male Right-handed9 10 Female Right-handed

假设我们想要根据性别和用手习惯对这段数据进行统计汇总。虽然可以用pivot_table实现该功能，但是pandas.crosstab函数会更方便：

In [151]: pd.crosstab(data.Gender, data.Handedness, margins=True)Out[151]:Handedness Left-handed Right-handed AllGenderFemale 1 4 5Male 2 3 5All 3 7 10

crosstab的前两个参数可以是数组、Series或数组列表。再比如对小费数据集：

In [152]: pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)Out[152]:smoker No Yes Alltime dayDinner Fri 3 9 12 Sat 45 42 87 Sun 57 19 76 Thur 1 0 1Lunch Fri 1 6 7 Thur 44 17 61All 151 93 244

示例：2012联邦选举委员会数据库

美国联邦选举委员会发布了有关政治竞选赞助方面的数据。其中包括赞助者的姓名、职业、雇主、地址以及出资额等信息。我们对2012年美国总统大选的数据集比较感兴趣（http://www.fec.gov/disclosurep/PDownload.do）。到编写本书时为止（2012年6月），涵盖全美各州的完整数据集是一个150MB的CSV文件（P00000001-ALL.csv），我们先用pandas.read_csv将其加载进来：

In [13]: fec = pd.read_csv('ch09/P00000001-ALL.csv')In [14]: fecOut[14]:Int64Index: 1001731 entries, 0 to 1001730Data columns:cmte_id 1001731 non-null valuescand_id 1001731 non-null valuescand_nm 1001731 non-null valuescontbr_nm 1001731 non-null valuescontbr_city 1001716 non-null valuescontbr_st 1001727 non-null valuescontbr_zip 1001620 non-null valuescontbr_employer 994314 non-null valuescontbr_occupation 994433 non-null valuescontb_receipt_amt 1001731 non-null valuescontb_receipt_dt 1001731 non-null valuesreceipt_desc 14166 non-null valuesmemo_cd 92482 non-null valuesmemo_text 97770 non-null valuesform_tp 1001731 non-null valuesfile_num 1001731 non-null valuesdtypes: float64(1), int64(1), object(14)

该DataFrame中的记录如下所示：

In [15]: fec.ix[123456]Out[15]:cmte_id C00431445cand_id P80003338cand_nm Obama, Barackcontbr_nm ELLMAN, IRAcontbr_city TEMPEcontbr_st AZcontbr_zip 852816719contbr_employer ARIZONA STATE UNIVERSITYcontbr_occupation PROFESSORcontb_receipt_amt 50contb_receipt_dt 01-DEC-11receipt_desc NaNmemo_cd NaNmemo_text NaNform_tp SA17Afile_num 772372Name: 123456

你可能已经想出了许多办法从这些竞选赞助数据中抽取有关赞助人和赞助模式的统计信息。我将在接下来的内容中介绍几种不同的分析工作（运用到目前为止已经学到的技术）。

不难看出，该数据中没有党派信息，因此最好把它加进去。通过unique，你可以获取全部的候选人名单（注意，NumPy不会输出信息中字符串两侧的引号）：

In [16]: unique_cands = fec.cand_nm.unique()In [17]: unique_candsOut[17]:array([Bachmann, Michelle, Romney, Mitt, Obama, Barack, Roemer, Charles E. 'Buddy' III, Pawlenty, Timothy, Johnson, Gary Earl, Paul, Ron, Santorum, Rick, Cain, Herman, Gingrich, Newt, McCotter, Thaddeus G, Huntsman, Jon, Perry, Rick], dtype=object)In [18]: unique_cands[2]Out[18]: 'Obama, Barack'

最简单的办法是利用字典说明党派关系

注1

：

parties = {'Bachmann, Michelle': 'Republican', 'Cain, Herman': 'Republican', 'Gingrich, Newt': 'Republican', 'Huntsman, Jon': 'Republican', 'Johnson, Gary Earl': 'Republican', 'McCotter, Thaddeus G': 'Republican', 'Obama, Barack': 'Democrat', 'Paul, Ron': 'Republican', 'Pawlenty, Timothy': 'Republican', 'Perry, Rick': 'Republican', "Roemer, Charles E. 'Buddy' III": 'Republican', 'Romney, Mitt': 'Republican', 'Santorum, Rick': 'Republican'}

现在，通过这个映射以及Series对象的map方法，你可以根据候选人姓名得到一组党派信息：

In [20]: fec.cand_nm[123456:123461]Out[20]:123456 Obama, Barack123457 Obama, Barack123458 Obama, Barack123459 Obama, Barack123460 Obama, BarackName: cand_nmIn [21]: fec.cand_nm[123456:123461].map(parties)Out[21]:123456 Democrat123457 Democrat123458 Democrat123459 Democrat123460 DemocratName: cand_nm# 将其添加为一个新列In [22]: fec['party'] = fec.cand_nm.map(parties)In [23]: fec['party'].value_counts()Out[23]:Democrat 593746Republican 407985

这里有两个需要注意的地方。第一，该数据既包括赞助也包括退款（负的出资额）：

In [24]: (fec.contb_receipt_amt > 0).value_counts()Out[24]:True 991475False 10256

为了简化分析过程，我限定该数据集只能有正的出资额：

In [25]: fec = fec[fec.contb_receipt_amt > 0]

由于Barack Obama和Mitt Romney是最主要的两名候选人，所以我还专门准备了一个子集，只包含针对他们两人的竞选活动的赞助信息：

In [26]: fec_mrbo = fec[fec.cand_nm.isin(['Obama, Barack', 'Romney, Mitt'])]

根据职业和雇主统计赞助信息

基于职业的赞助信息统计是另一种经常被研究的统计任务。例如，律师们更倾向于资助民主党，而企业主则更倾向于资助共和党。你可以不相信我，自己看那些数据就知道了。首先，根据职业计算出资总额，这很简单：

In [27]: fec.contbr_occupation.value_counts()[:10]Out[27]:RETIRED 233990INFORMATION REQUESTED 35107ATTORNEY 34286HOMEMAKER 29931PHYSICIAN 23432INFORMATION REQUESTED PER BEST EFFORTS 21138ENGINEER 14334TEACHER 13990CONSULTANT 13273PROFESSOR 12555

不难看出，许多职业都涉及相同的基本工作类型，或者同一样东西有多种变体。下面的代码片段可以清理一些这样的数据（将一个职业信息映射到另一个）。注意，这里巧妙地利用了dict.get，它允许没有映射关系的职业也能"通过"：

occ_mapping = { 'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED', 'INFORMATION REQUESTED' : 'NOT PROVIDED', 'INFORMATION REQUESTED (BEST EFFORTS)' : 'NOT PROVIDED', 'C.E.O.': 'CEO'}# 如果没有提供相关映射，则返回xf = lambda x: occ_mapping.get(x, x)fec.contbr_occupation = fec.contbr_occupation.map(f)

我对雇主信息也进行了同样的处理：

emp_mapping = { 'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED', 'INFORMATION REQUESTED' : 'NOT PROVIDED', 'SELF' : 'SELF-EMPLOYED', 'SELF EMPLOYED' : 'SELF-EMPLOYED',}# 如果没有提供相关映射，则返回xf = lambda x: emp_mapping.get(x, x)fec.contbr_employer = fec.contbr_employer.map(f)

现在，你可以通过pivot_table根据党派和职业对数据进行聚合，然后过滤掉总出资额不足200万美元的数据：

In [34]: by_occupation = fec.pivot_table('contb_receipt_amt', ...: rows='contbr_occupation', ...: cols='party', aggfunc='sum')In [35]: over_2mm = by_occupation[by_occupation.sum(1) > 2000000]In [36]: over_2mmOut[36]:party Democrat Republicancontbr_occupationATTORNEY 11141982.97 7477194.430000CEO 2074974.79 4211040.520000CONSULTANT 2459912.71 2544725.450000ENGINEER 951525.55 1818373.700000EXECUTIVE 1355161.05 4138850.090000HOMEMAKER 4248875.80 13634275.780000INVESTOR 884133.00 2431768.920000LAWYER 3160478.87 391224.320000MANAGER 762883.22 1444532.370000NOT PROVIDED 4866973.96 20565473.010000OWNER 1001567.36 2408286.920000PHYSICIAN 3735124.94 3594320.240000PRESIDENT 1878509.95 4720923.760000PROFESSOR 2165071.08 296702.730000REAL ESTATE 528902.09 1625902.250000RETIRED 25305116.38 23561244.489999SELF-EMPLOYED 672393.40 1640252.540000

把这些数据做成柱状图看起来会更加清楚（'barh'表示水平柱状图，如图9-2所示）：

In [38]: over_2mm.plot(kind='barh')

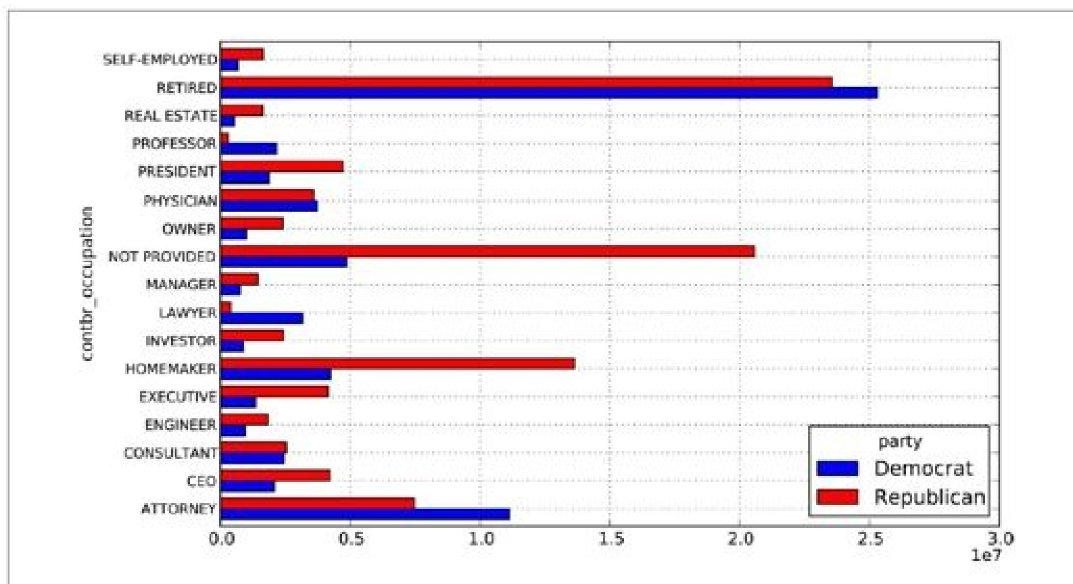


图9-2：对各党派总出资额最高的职业

你可能还想了解一下对Obama和Romney总出资额最高的职业和企业。为此，我们先对候选人进行分组，然后使用本章前面介绍的那种求取最大值的方法：

def get_top_amounts(group, key, n=5):   totals = group.groupby(key)['contb_receipt_amt'].sum() # 根据key对totals进行降序排列 return totals.order(ascending=False)[n:]

然后根据职业和雇主进行聚合：

In [40]: grouped = fec_mrbo.groupby('cand_nm')In [41]: grouped.apply(get_top_amounts, 'contbr_occupation', n=7)Out[41]:cand_nm contbr_occupationObama, Barack RETIRED 25305116.38 ATTORNEY 11141982.97 INFORMATION REQUESTED 4866973.96 HOMEMAKER 4248875.80 PHYSICIAN 3735124.94 LAWYER 3160478.87 CONSULTANT 2459912.71Romney, Mitt RETIRED 11508473.59 INFORMATION REQUESTED PER BEST EFFORTS 11396894.84 HOMEMAKER 8147446.22 ATTORNEY 5364718.82 PRESIDENT 2491244.89 EXECUTIVE 2300947.03 C.E.O. 1968386.11Name: contb_receipt_amtIn [42]: grouped.apply(get_top_amounts, 'contbr_employer', n=10)Out[42]:cand_nm contbr_employerObama, Barack RETIRED 22694358.85 SELF-EMPLOYED 17080985.96 NOT EMPLOYED 8586308.70 INFORMATION REQUESTED 5053480.37 HOMEMAKER 2605408.54 SELF 1076531.20 SELF EMPLOYED 469290.00 STUDENT 318831.45 VOLUNTEER 257104.00 MICROSOFT 215585.36Romney, Mitt INFORMATION REQUESTED PER BEST EFFORTS 12059527.24 RETIRED 11506225.71 HOMEMAKER 8147196.22 SELF-EMPLOYED 7409860.98 STUDENT 496490.94 CREDIT SUISSE 281150.00 MORGAN STANLEY 267266.00 GOLDMAN SACH & CO. 238250.00 BARCLAYS CAPITAL 162750.00 H.I.G. CAPITAL 139500.00Name: contb_receipt_amt

对出资额分组

还可以对该数据做另一种非常实用的分析：利用cut函数根据出资额的大小将数据离散化到多个面元中：

In [43]: bins = np.array([0, 1, 10, 100, 1000, 10000, 100000, 1000000, 10000000])In [44]: labels = pd.cut(fec_mrbo.contb_receipt_amt, bins)In [45]: labelsOut[45]:Categorical: contb_receipt_amtarray([(10, 100], (100, 1000], (100, 1000], ..., (1, 10], (10, 100], (100, 1000]], dtype=object)Levels (8): Index([(0, 1], (1, 10], (10, 100], (100, 1000], (1000, 10000],   (10000, 100000], (100000, 1000000], (1000000, 10000000]], dtype=object)

然后根据候选人姓名以及面元标签对数据进行分组：

In [46]: grouped = fec_mrbo.groupby(['cand_nm', labels])In [47]: grouped.size().unstack(0)Out[47]:cand_nm Obama, Barack Romney, Mittcontb_receipt_amt(0, 1] 493 77(1, 10] 40070 3681(10, 100] 372280 31853(100, 1000] 153991 43357(1000, 10000] 22284 26186(10000, 100000] 2 1(100000, 1000000] 3 NaN(1000000, 10000000] 4 NaN

从这个数据中可以看出，在小额赞助方面，Obama获得的数量比Romney多得多。你还可以对出资额求和并在面元内规格化，以便图形化显示两位候选人各种赞助额度的比例：

In [48]: bucket_sums = grouped.contb_receipt_amt.sum().unstack(0)In [49]: bucket_sumsOut[49]:cand_nm Obama, Barack Romney, Mittcontb_receipt_amt(0, 1] 318.24 77.00(1, 10] 337267.62 29819.66(10, 100] 20288981.41 1987783.76(100, 1000] 54798531.46 22363381.69(1000, 10000] 51753705.67 63942145.42(10000, 100000] 59100.00 12700.00(100000, 1000000] 1490683.08 NaN(1000000, 10000000] 7148839.76 NaNIn [50]: normed_sums = bucket_sums.div(bucket_sums.sum(axis=1), axis=0)In [51]: normed_sumsOut[51]:cand_nm Obama, Barack Romney, Mittcontb_receipt_amt(0, 1] 0.805182 0.194818(1, 10] 0.918767 0.081233(10, 100] 0.910769 0.089231(100, 1000] 0.710176 0.289824(1000, 10000] 0.447326 0.552674(10000, 100000] 0.823120 0.176880(100000, 1000000] 1.000000 NaN(1000000, 10000000] 1.000000 NaNIn [52]: normed_sums[:-2].plot(kind='barh', stacked=True)

我排除了两个最大的面元，因为这些不是由个人捐赠的。最终的结果如图9-3所示。

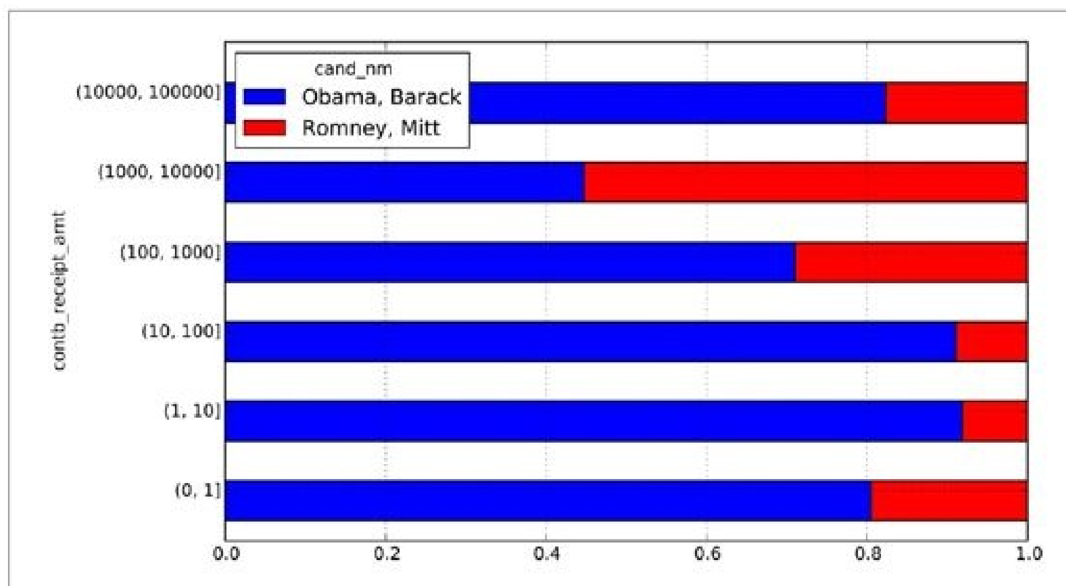


图9-3：两位候选人收到的各种捐赠额度的总额比例

当然，还可以对该分析过程做许多的提炼和改进。比如说，可以根据赞助人的姓名和邮编对数据进行聚合，以便找出哪些人进行了多次小额捐款，哪些人又进行了一次或多次大额捐款。我强烈建议你下载这些数据并自己摸索一下。

根据州统计赞助信息

首先自然是根据候选人和州对数据进行聚合：

In [53]: grouped = fec_mrbo.groupby(['cand_nm', 'contbr_st'])In [54]: totals = grouped.contb_receipt_amt.sum().unstack(0).fillna(0)In [55]: totals = totals[totals.sum(1) > 100000]In [56]: totals[:10]Out[56]:cand_nm Obama, Barack Romney, Mittcontbr_stAK 281840.15 86204.24AL 543123.48 527303.51AR 359247.28 105556.00AZ 1506476.98 1888436.23CA 23824984.24 11237636.60CO 2132429.49 1506714.12CT 2068291.26 3499475.45DC 4373538.80 1025137.50DE 336669.14 82712.00FL 7318178.58 8338458.81

如果对各行除以总赞助额，就会得到各候选人在各州的总赞助额比例：

In [57]: percent = totals.div(totals.sum(1), axis=0)In [58]: percent[:10]Out[58]:cand_nm Obama, Barack Romney, Mittcontbr_stAK 0.765778 0.234222AL 0.507390 0.492610AR 0.772902 0.227098AZ 0.443745 0.556255CA 0.679498 0.320502CO 0.585970 0.414030CT 0.371476 0.628524DC 0.810113 0.189887DE 0.802776 0.197224FL 0.467417 0.532583

我认为在地图上看这些数据会比较有意思（第8章中介绍过相关技术）。在找到有关州界的shape file（http://nationalatlas.gov/atlasftp.html?openChapters=chpbound）并稍微学习一下matplotlib及其basemap工具包（Thomas Lecocq的博客帮了我的大忙

注2

）之后，我终于用下面这段代码画出了刚才算出来的相对百分比：

译注7

from mpl_toolkits.basemap import Basemap, cmimport numpy as npfrom matplotlib import rcParamsfrom matplotlib.collections import LineCollectionimport matplotlib.pyplot as pltfrom shapelib import ShapeFileimport dbflibobama = percent['Obama, Barack']fig = plt.figure(figsize=(12, 12))ax = fig.add_axes([0.1,0.1,0.8,0.8])lllat = 21; urlat = 53; lllon = -118; urlon = -62m = Basemap(ax=ax, projection='stere', lon_0=(urlon + lllon) / 2, lat_0=(urlat + lllat) / 2, llcrnrlat=lllat, urcrnrlat=urlat, llcrnrlon=lllon, urcrnrlon=urlon, resolution='l')m.drawcoastlines()m.drawcountries()shp = ShapeFile('../states/statesp020')dbf = dbflib.open('../states/statesp020')for npoly in range(shp.info()[0]): # 在地图上绘制彩色多边形 shpsegs = [] shp_object = shp.read_object(npoly) verts = shp_object.vertices() rings = len(verts) for ring in range(rings): lons, lats = zip(*verts[ring]) x, y = m(lons, lats) shpsegs.append(zip(x,y)) if ring == 0: shapedict = dbf.read_record(npoly) name = shapedict['STATE'] lines = LineCollection(shpsegs,antialiaseds=(1,)) # state_to_code字典，例如'ALASKA' -> 'AK', omitted try: per = obama[state_to_code[name.upper()]] except KeyError: continue lines.set_facecolors('k') lines.set_alpha(0.75 * per) # 把“百分比”变小一点 lines.set_edgecolors('k') lines.set_linewidth(0.3)plt.show()

最终结果如图9-4所示。

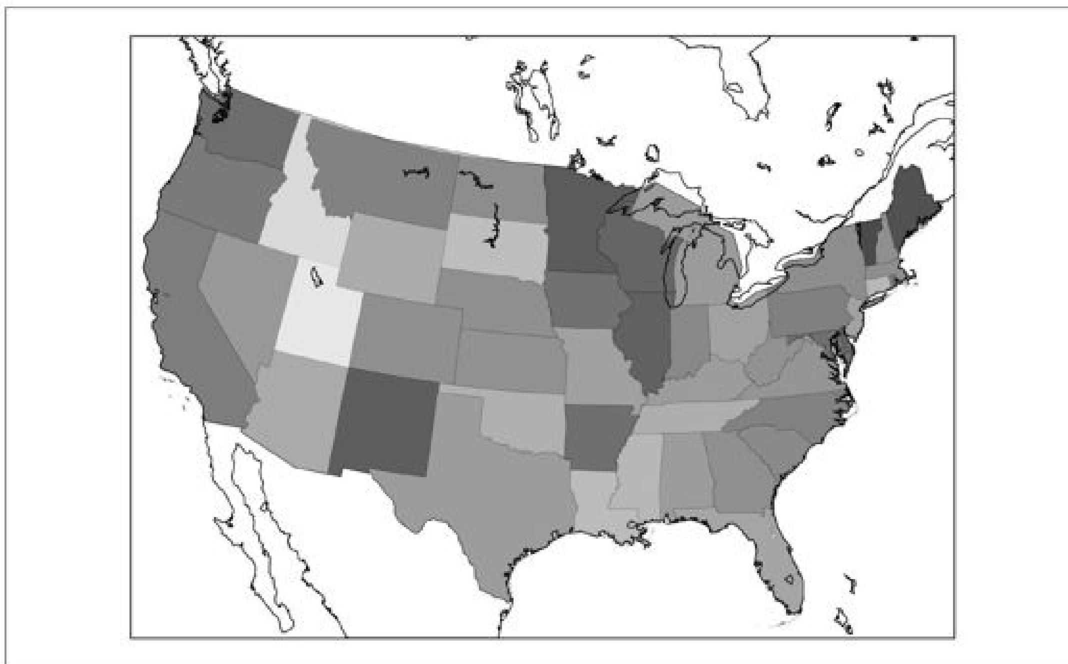


图9-4：汇集了所有赞助统计信息的美国地图（颜色越深表示越支持民主党）

注1

：为了简单起见，这里假设Gary Johnson是一名共和党员，虽然他后来成为自由党的候选人。

注2

：http://www.geophysique.be/2011/01/27/matplotlib-basemap-tutorial-07-shapefiles-unleached/。

译注7

：惭愧，折腾了整两天，愣是没做出来。太郁闷了，照着输入都不行。在网上找了一个比较有效的办法，不过由于时间太紧就没完成，希望读者在尝试成功之后一定在网上发布一下，以飨更多读者。

第10章　时间序列

不管在哪个领域中（如金融学、经济学、生态学、神经科学、物理学等），时间序列（time series）数据都是一种重要的结构化数据形式。在多个时间点观察或测量到的任何事物可以形成一段时间序列。很多时间序列是固定频率的，也就是说，数据点是根据某种规律定期出现的（比如每15秒、每5分钟、每月出现一次）。时间序列也可以是不定期的。时间序列数据的意义取决于具体的应用场景，主要有以下几种：

·时间戳（timestamp），特定的时刻。

·固定时期（period），如2007年1月或2010年全年。

·时间间隔（interval），由起始和结束时间戳表示。时期（period）可以被看做间隔（interval）的特例。

·实验或过程时间，每个时间点都是相对于特定起始时间的一个度量。例如，从放入烤箱时起，每秒钟饼干的直径。

本章主要讲解前3种时间序列。许多技术都可用于处理实验型时间序列，其索引可能是一个整数或浮点数（表示从实验开始算起已经过去的时间）。最简单也最常见的时间序列都是用时间戳进行索引的。

pandas提供了一组标准的时间序列处理工具和数据算法。因此，你可以高效处理非常大的时间序列，轻松地进行切片/切块、聚合、对定期/不定期的时间序列进行重采样等。可能你已经猜到了，这些工具中大部分都对金融和经济数据尤为有用，但你当然也可以用它们来分析服务器日志数据。

注意：本章中部分功能和代码（比如处理时期的那些）用到了已经停止更新的scikits.timeseries库。译注1

译注1

：没找到2.7的，但是网上好像有人用了。

日期和时间数据类型及工具

Python标准库包含用于日期（date）和时间（time）数据的数据类型，而且还有日历方面的功能。我们主要会用到datetime、time以及calendar模块。datetime.datetime（也可以简写为datetime）是用得最多的数据类型：

In [317]: from datetime import datetimeIn [318]: now = datetime.now()In [319]: nowOut[319]: datetime.datetime(2012, 8, 4, 17, 9, 21, 832092)In [320]: now.year, now.month, now.dayOut[320]: (2012, 8, 4)

datetime以毫秒形式存储日期和时间。datetime.timedelta表示两个datetime对象之间的时间差：

In [321]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)In [322]: deltaOut[322]: datetime.timedelta(926, 56700)In [323]: delta.daysOut[323]: 926In [324]: delta.secondsOut[324]: 56700

可以给datetime对象加上（或减去）一个或多个timedelta，这样会产生一个新对象：

In [325]: from datetime import timedeltaIn [326]: start = datetime(2011, 1, 7)In [327]: start + timedelta(12)Out[327]: datetime.datetime(2011, 1, 19, 0, 0)In [328]: start - 2 * timedelta(12)Out[328]: datetime.datetime(2010, 12, 14, 0, 0)

datetime模块中的数据类型参见表10-1。虽然本章主要讲的是pandas数据类型和高级时间序列处理，但你肯定会在Python的其他地方遇到有关datetime的数据类型。

表10-1：datetime模块中的数据类型

类型	说明
date	以公历形式存储日历日期（年、月、日）
time	将时间存储为时、分、秒、毫秒
datetime	存储日期和时间
timedelta	表示两个datetime值之间的差（日、秒、毫秒）

字符串和datetime的相互转换

利用str或strftime方法（传入一个格式化字符串），datetime对象和pandas的Timestamp对象（稍后就会介绍）可以被格式化为字符串：

In [329]: stamp = datetime(2011, 1, 3)In [330]: str(stamp) In [331]: stamp.strftime('%Y-%m-%d')Out[330]: '2011-01-03 00:00:00' Out[331]: '2011-01-03'

表10-2列出了全部的格式化编码。datetime.strptime也可以用这些格式化编码将字符串转换为日期：

In [332]: value = '2011-01-03'In [333]: datetime.strptime(value, '%Y-%m-%d')Out[333]: datetime.datetime(2011, 1, 3, 0, 0)In [334]: datestrs = ['7/6/2011', '8/6/2011']In [335]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]Out[335]: [datetime.datetime(2011, 7, 6, 0, 0), datetime.datetime(2011, 8, 6, 0, 0)]

datetime.strptime是通过已知格式进行日期解析的最佳方式。但是每次都要编写格式定义是很麻烦的事情，尤其是对于一些常见的日期格式。这种情况下，你可以用dateutil这个第三方包中的parser.parse方法：

In [336]: from dateutil.parser import parseIn [337]: parse('2011-01-03')Out[337]: datetime.datetime(2011, 1, 3, 0, 0)

dateutil可以解析几乎所有人类能够理解的日期表示形式

译注2

：

In [338]: parse('Jan 31, 1997 10:45 PM')Out[338]: datetime.datetime(1997, 1, 31, 22, 45)

在国通用的格式中，日通常出现在月的前面，传入dayfirst=True即可解决这个问题：

In [339]: parse('6/12/2011', dayfirst=True)Out[339]: datetime.datetime(2011, 12, 6, 0, 0)

pandas通常是用于处理成组日期的，不管这些日期是DataFrame的轴索引还是列。to_datetime方法可以解析多种不同的日期表示形式。对标准日期格式（如ISO8601）的解析非常快。

In [340]: datestrsOut[340]: ['7/6/2011', '8/6/2011']In [341]: pd.to_datetime(datestrs)Out[341]: [2011-07-06 00:00:00, 2011-08-06 00:00:00]Length: 2, Freq: None, Timezone: None

它还可以处理缺失值（None、空字符串等）：

In [342]: idx = pd.to_datetime(datestrs + [None])In [343]: idxOut[343]:[2011-07-06 00:00:00, ..., NaT]Length: 3, Freq: None, Timezone: NoneIn [344]: idx[2]Out[344]: NaTIn [345]: pd.isnull(idx)Out[345]: array([False, False, True], dtype=bool)

NaT（Not a Time）是pandas中时间戳数据的NA值。

警告：dateutil.parser是一个实用但不完美的工具。比如说，它会把一些原本不是日期的字符串认作是日期（比如"42"会被解析为2042年的今天）。

表10-2：datetime格式定义（兼容ISO C89）

代码	说明
%Y	4位数的年
%y	2位数的年
%m	2位数的月[01, 12]
%d	2位数的日[01, 31]

表10-2：datetime格式定义（兼容ISO C89）（续）

代码	说明
%H	时（24小时制）[00, 23]
%I	时（12小时制）[01, 12]
%M	2位数的分[00, 59]
%S	秒[00, 61]（秒60和61用于闰秒）
%w	用整数表示的星期几[0（星期天），6]
%U	每年的第几周[00, 53]。星期天被认为是每周的第一天，每年第一个星期天之前的那几天被认为是“第0周”
%W	每年的第几周[00, 53]。星期一被认为是每周的第一天，每年第一个星期一之前的那几天被认为是“第0周”
%z	以+HHMM或-HHMM表示的UTC时区偏移量，如果时区为naive[译注3]，则返回空字符串
%F	%Y-%m-%d简写形式，例如2012-4-18[译注4]
%D	%m/%d/%y简写形式，例如04/18/12

译注3：更准确一点地讲，应该是时间对象，而不是时区。时间对象有naive和aware之分，简单地说，就是有没有人为调整（比如夏令时之类的东西）。

译注4：应该是2012-04-18才对。

datetime对象还有一些特定于当前环境（位于不同国家或使用不同语言的系统）的格式化选项。例如，德语或法语系统所用的月份简写就与英语系统所用的不同。

表10-3：特定于当前环境的日期格式

代码	说明
%a	星期几的简写
%A	星期几的全称
%b	月份的简写
%B	月份的全称
%c	完整的日期和时间，例如“Tue 01 May 2012 04:20:57 PM”
%p	不同环境中的AM或PM
%x	适合于当前环境的日期格式，例如，在美国，“May 1, 2012”会产生“05/01/2012”
%X	适合于当前环境的时间格式，例如“04:24:12 PM”

译注2

：很遗憾，中文不行。

时间序列基础

pandas最基本的时间序列类型就是以时间戳（通常以Python字符串或datatime对象表示）为索引的Series：

In [346]: from datetime import datetimeIn [347]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5), datetime(2011, 1, 7), ...: datetime(2011, 1, 8), datetime(2011, 1, 10), datetime(2011, 1, 12)]In [348]: ts = Series(np.random.randn(6), index=dates)In [349]: tsOut[349]:2011-01-02 0.6900022011-01-05 1.0015432011-01-07 -0.5030872011-01-08 -0.6222742011-01-10 -0.9211692011-01-12 -0.726213

这些datetime对象实际上是被放在一个DatetimeIndex中的。现在，变量ts就成为一个TimeSeries了：

In [350]: type(ts)Out[350]: pandas.core.series.TimeSeriesIn [351]: ts.indexOut[351]:[2011-01-02 00:00:00, ..., 2011-01-12 00:00:00]Length: 6, Freq: None, Timezone: None

注意：没必要显式使用TimeSeries的构造函数。当创建一个带有DatetimeIndex的Series时，pandas就会知道该对象是一个时间序列。

跟其他Series一样，不同索引的时间序列之间的算术运算会自动按日期对齐：

In [352]: ts + ts[::2]Out[352]:2011-01-02 1.3800042011-01-05 NaN2011-01-07 -1.0061752011-01-08 NaN2011-01-10 -1.8423372011-01-12 NaN

pandas用NumPy的datetime64数据类型以纳秒形式存储时间戳：

In [353]: ts.index.dtypeOut[353]: dtype('datetime64[ns]')

DatetimeIndex中的各个标量值是pandas的Timestamp对象：

In [354]: stamp = ts.index[0]In [355]: stampOut[355]:

只要有需要，TimeStamp可以随时自动转换为datetime对象。此外，它还可以存储频率信息（如果有的话），且知道如何执行时区转换以及其他操作。稍后将对此进行详细讲解。

索引、选取、子集构造

由于TimeSeries是Series的一个子类，所以在索引以及数据选取方面它们的行为是一样的：

In [356]: stamp = ts.index[2]In [357]: ts[stamp]Out[357]: -0.50308739136034464

还有一种更为方便的用法：传入一个可以被解释为日期的字符串。

In [358]: ts['1/10/2011']Out[358]: -0.92116860801301081In [359]: ts['20110110']Out[359]: -0.92116860801301081

对于较长的时间序列，只需传入“年”或“年月”即可轻松选取数据的切片：

In [360]: longer_ts = Series(np.random.randn(1000), ...: index=pd.date_range('1/1/2000', periods=1000))In [361]: longer_tsOut[361]:2000-01-01 0.2228962000-01-02 0.0513162000-01-03 -1.1577192000-01-04 0.816707...2002-09-23 -0.3958132002-09-24 -0.1807372002-09-25 1.3375082002-09-26 -0.416584Freq: D, Length: 1000In [362]:

longer_ts['2001']Out[362]:2001-01-01 -1.4995032001-01-02 0.5451542001-01-03 0.4008232001-01-04 -1.946230...2001-12-28 -1.5681392001-12-29 -0.9008872001-12-30 0.6523462001-12-31 0.871600Freq: D, Length: 365In [363]: longer_ts['2001-05']Out[363]:2001-05-01 1.6620142001-05-02 -1.1892032001-05-03 0.0935972001-05-04 -0.539164...2001-05-28 -0.6830662001-05-29 -0.9503132001-05-30 0.4007102001-05-31 -0.126072Freq: D, Length: 31

通过日期进行切片的方式只对规则Series有效：

In [364]: ts[datetime(2011, 1, 7):]Out[364]:2011-01-07 -0.5030872011-01-08 -0.6222742011-01-10 -0.9211692011-01-12 -0.726213

由于大部分时间序列数据都是按照时间先后排序的，因此你也可以用不存在于该时间序列中的时间戳对其进行切片（即范围查询）：

In [365]: tsOut[365]:2011-01-02 0.6900022011-01-05 1.0015432011-01-07 -0.5030872011-01-08 -0.6222742011-01-10 -0.9211692011-01-12 -0.726213In [366]: ts['1/6/2011':'1/11/2011']Out[366]:2011-01-07 -0.5030872011-01-08 -0.6222742011-01-10 -0.921169

跟之前一样，这里可以传入字符串日期、datetime或Timestamp。注意，这样切片所产生的是源时间序列的视图，跟NumPy数组的切片运算是一样的。此外，还有一个等价的实例方法也可以截取两个日期之间TimeSeries：

In [367]: ts.truncate(after='1/9/2011')Out[367]:2011-01-02 0.6900022011-01-05 1.0015432011-01-07 -0.5030872011-01-08 -0.622274

上面这些操作对DataFrame也有效。例如，对DataFrame的行进行索引：

In [368]: dates = pd.date_range('1/1/2000', periods=100, freq='W-WED')In [369]: long_df = DataFrame(np.random.randn(100, 4), ...: index=dates, ...: columns=['Colorado', 'Texas', 'New York', 'Ohio'])In [370]: long_df.ix['5-2001']Out[370]: Colorado Texas New York Ohio2001-05-02 0.943479 -0.349366 0.530412 -0.5087242001-05-09 0.230643 -0.065569 -0.248717 -0.5871362001-05-16 -1.022324 1.060661 0.954768 -0.5118242001-05-23 -1.387680 0.767902 -1.164490 1.5270702001-05-30 0.287542 0.715359 -0.345805 0.470886

带有重复索引的时间序列

在某些应用场景中，可能会存在多个观测数据落在同一个时间点上的情况。下面就是一个例子：

In [371]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/2/2000', ...: '1/3/2000'])In [372]: dup_ts = Series(np.arange(5), index=dates)In [373]: dup_tsOut[373]:2000-01-01 02000-01-02 12000-01-02 22000-01-02 32000-01-03 4

通过检查索引的is_unique属性，我们就可以知道它是不是唯一的：

In [374]: dup_ts.index.is_uniqueOut[374]: False

对这个时间序列进行索引，要么产生标量值，要么产生切片，具体要看所选的时间点是否重复：

In [375]: dup_ts['1/3/2000'] # 不重复Out[375]: 4In [376]: dup_ts['1/2/2000'] # 重复Out[376]:2000-01-02 12000-01-02 22000-01-02 3

假设你想要对具有非唯一时间戳的数据进行聚合。一个办法是使用groupby，并传入level=0（索引的唯一一层！）：

In [377]: grouped = dup_ts.groupby(level=0)In [378]: grouped.mean() In [379]: grouped.count()Out[378]: Out[379]:2000-01-01 0 2000-01-01 12000-01-02 2 2000-01-02 32000-01-03 4 2000-01-03 1

日期的范围、频率以及移动

pandas中的时间序列一般被认为是不规则的，也就是说，它们没有固定的频率。对于大部分应用程序而言，这是无所谓的。但是，它常常需要以某种相对固定的频率进行分析，比如每日、每月、每15分钟等（这样自然会在时间序列中引入缺失值）。幸运的是，pandas有一整套标准时间序列频率以及用于重采样、频率推断、生成固定频率日期范围的工具。例如，我们可以将之前那个时间序列转换为一个具有固定频率（每日）的时间序列，只需调用resample即可：

In [380]: ts In [381]: ts.resample('D')Out[380]: Out[381]:2011-01-02 0.690002 2011-01-02 0.6900022011-01-05 1.001543 2011-01-03 NaN2011-01-07 -0.503087 2011-01-04 NaN2011-01-08 -0.622274 2011-01-05 1.0015432011-01-10 -0.921169 2011-01-06 NaN2011-01-12 -0.726213 2011-01-07 -0.503087 2011-01-08 -0.622274 2011-01-09 NaN 2011-01-10 -0.921169 2011-01-11 NaN 2011-01-12 -0.726213 Freq: D

频率的转换（或重采样）是一个比较大的主题，稍后将专门用一节来进行讨论。这里我将告诉你如何使用基本的频率。

生成日期范围

虽然我之前用的时候没有明说，但你可能已经猜到pandas.date_range可用于生成指定长度的DatetimeIndex：

In [382]: index = pd.date_range('4/1/2012', '6/1/2012')In [383]: indexOut[383]:[2012-04-01 00:00:00, ..., 2012-06-01 00:00:00]Length: 62, Freq: D, Timezone: None

默认情况下，date_range会产生按天计算的时间点。如果只传入起始或结束日期，那就还得传入一个表示一段时间的数字：

In [384]: pd.date_range(start='4/1/2012', periods=20)Out[384]:[2012-04-01 00:00:00, ..., 2012-04-20 00:00:00]Length: 20, Freq: D, Timezone: NoneIn [385]: pd.date_range(end='6/1/2012', periods=20)Out[385]:[2012-05-13 00:00:00, ..., 2012-06-01 00:00:00]Length: 20, Freq: D, Timezone: None

起始和结束日期定义了日期索引的严格边界。例如，如果你想要生成一个由每月最后一个工作日组成的日期索引，可以传入"BM"频率（表示business end of month），这样就只会包含时间间隔内（或刚好在边界上的）符合频率要求的日期：

In [386]: pd.date_range('1/1/2000', '12/1/2000', freq='BM')Out[386]:[2000-01-31 00:00:00, ..., 2000-11-30 00:00:00]Length: 11, Freq: BM, Timezone: None

date_range默认会保留起始和结束时间戳的时间信息（如果有的话）：

In [387]: pd.date_range('5/2/2012 12:56:31', periods=5)Out[387]:[2012-05-02 12:56:31, ..., 2012-05-06 12:56:31]Length: 5, Freq: D, Timezone: None

有时，虽然起始和结束日期带有时间信息，但你希望产生一组被规范化（normalize）到午夜的时间戳。normalize选项即可实现该功能：

In [388]: pd.date_range('5/2/2012 12:56:31', periods=5, normalize=True)Out[388]:[2012-05-02 00:00:00, ..., 2012-05-06 00:00:00]Length: 5, Freq: D, Timezone: None

频率和日期偏移量

pandas中的频率是由一个基础频率（base frequency）和一个乘数组成的。基础频率通常以一个字符串别名表示，比如"M"表示每月，"H"表示每小时。对于每个基础频率，都有一个被称为日期偏移量（date offset）的对象与之对应。例如，按小时计算的频率可以用Hour类表示：

In [389]: from pandas.tseries.offsets import Hour, MinuteIn [390]: hour = Hour()In [391]: hourOut[391]: <1 Hour="">

传入一个整数即可定义偏移量的倍数：

In [392]: four_hours = Hour(4)In [393]: four_hoursOut[393]: <4 Hours="">

一般来说，无需显式创建这样的对象，只需使用诸如"H"或"4H"这样的字符串别名即可。在基础频率前面放上一个整数即可创建倍数：

In [394]: pd.date_range('1/1/2000', '1/3/2000 23:59', freq='4h')Out[394]:[2000-01-01 00:00:00, ..., 2000-01-03 20:00:00]Length: 18, Freq: 4H, Timezone: None

大部分偏移量对象都可通过加法进行连接：

In [395]: Hour(2) + Minute(30)Out[395]: <150 Minutes="">

同理，你也可以传入频率字符串（如"2h30min"），这种字符串可以被高效地解析为等效的表达式：

In [396]: pd.date_range('1/1/2000', periods=10, freq='1h30min')Out[396]:[2000-01-01 00:00:00, ..., 2000-01-01 13:30:00]Length: 10, Freq: 90T, Timezone: None

有些频率所描述的时间点并不是均匀分隔的。例如，"M"（日历月末）和"BM"（每月最后一个工作日）就取决于每月的天数，对于后者，还要考虑月末是不是周末。由于没有更好的术语，我将这些称为锚点偏移量（anchored offset）。

表10-4列出了pandas中的频率代码和日期偏移量类。

注意：用户可以根据实际需求自定义一些频率类以便提供pandas所没有的日期逻辑，但具体的细节超出了本书的范围。

表10-4：时间序列的基础频率

别名	偏移量类型	说明
D	Day	每日历日
B	BusinessDay	每工作日
H	Hour	每小时
T或min	Minute	每分
S	Second	每秒
L或ms	Milli	每毫秒（即每千分之一秒）
U	Micro	每微秒（即每百万分之一秒）
M	MonthEnd	每月最后一个日历日
BM	BusinessMonthEnd	每月最后一个工作日
MS	MonthBegin	每月第一个日历日
BMS	BusinessMonthBegin	每月第一个工作日
W-MON、W-TUE…	Week	从指定的星期几（MON、TUE、WED、THU、FRI、SAT、SUN）开始算起，每周
WOM-1MON、WOM-2MON…	WeekOfMonth	产生每月第一、第二、第三或第四周的星期几。例如，WOM-3FRI表示每月第3个星期五
Q-JAN、Q-FEB…	QuarterEnd	对于以指定月份（JAN、FEB、MAR、APR、MAY、JUN、JUL、AUG、SEP、OCT、NOV、DEC）结束的年度，每季度最后一月的最后一个日历日
BQ-JAN、BQ-FEB…	BusinessQuarterEnd	对于以指定月份结束的年度，每季度最后一月的最后一个工作日

表10-4：时间序列的基础频率（续）

别名	偏移量类型	说明
QS-JAN、QS-FEB…	QuarterBegin	对于以指定月份结束的年度，每季度最后一月的第一个日历日
BQS-JAN、BQS-FEB…	BusinessQuarterBegin	对于以指定月份结束的年度，每季度最后一月的第一个工作日
A-JAN、A-FEB…	YearEnd	每年指定月份（JAN、FEB、MAR、APR、MAY、JUN、JUL、AUG、SEP、OCT、NOV、DEC）的最后一个日历日
BA-JAN、BA-FEB…	BusinessYearEnd	每年指定月份的最后一个工作日
AS-JAN、AS-FEB…	YearBegin	每年指定月份的第一个日历日
BAS-JAN、BAS-FEB…	BusinessYearBegin	每年指定月份的第一个工作日

WOM日期

WOM（Week Of Month）是一种非常实用的频率类，它以WOM开头。它使你能获得诸如“每月第3个星期五”之类的日期：

In [397]: rng = pd.date_range('1/1/2012', '9/1/2012', freq='WOM-3FRI')In [398]: list(rng)Out[398]:[, , , , , , ,]

美国的股票期权交易人会意识到这些日子就是标准的月度到期日。

移动（超前和滞后）数据

移动（shifting）指的是沿着时间轴将数据前移或后移。Series和DataFrame都有一个shift方法用于执行单纯的前移或后移操作，保持索引不变：

In [399]: ts = Series(np.random.randn(4), ...: index=pd.date_range('1/1/2000', periods=4, freq='M'))In [400]: ts In [401]: ts.shift(2) In [402]: ts.shift(-2)Out[400]: Out[401]: Out[402]:2000-01-31 0.575283 2000-01-31 NaN 2000-01-31 1.8145822000-02-29 0.304205 2000-02-29 NaN 2000-02-29 1.6348582000-03-31 1.814582 2000-03-31 0.575283 2000-03-31 NaN2000-04-30 1.634858 2000-04-30 0.304205 2000-04-30 NaNFreq: M Freq: M Freq: M

shift通常用于计算一个时间序列或多个时间序列（如DataFrame的列）中的百分比变化。可以这样表达：

ts / ts.shift(1) - 1

由于单纯的移位操作不会修改索引，所以部分数据会被丢弃。因此，如果频率已知，则可以将其传给shift以便实现对时间戳进行位移而不是对数据进行简单位移：

In [403]: ts.shift(2, freq='M')Out[403]:2000-03-31 0.5752832000-04-30 0.3042052000-05-31 1.8145822000-06-30 1.634858Freq: M

这里还可以使用其他频率，于是你就能非常灵活地对数据进行超前和滞后处理了：

In [404]: ts.shift(3, freq='D') In [405]: ts.shift(1, freq='3D')Out[404]: Out[405]:2000-02-03 0.575283 2000-02-03 0.5752832000-03-03 0.304205 2000-03-03 0.3042052000-04-03 1.814582 2000-04-03 1.8145822000-05-03 1.634858 2000-05-03 1.634858In [406]: ts.shift(1, freq='90T')Out[406]:2000-01-31 01:30:00 0.5752832000-02-29 01:30:00 0.3042052000-03-31 01:30:00 1.8145822000-04-30 01:30:00 1.634858

通过偏移量对日期进行位移

pandas的日期偏移量还可以用在datetime或Timestamp对象上：

In [407]: from pandas.tseries.offsets import Day, MonthEndIn [408]: now = datetime(2011, 11, 17)In [409]: now + 3 * Day()Out[409]: datetime.datetime(2011, 11, 20, 0, 0)

如果加的是锚点偏移量（比如MonthEnd），第一次增量会将原日期向前滚动到符合频率规则的下一个日期译注5：

In [410]: now + MonthEnd()Out[410]: datetime.datetime(2011, 11, 30, 0, 0)In [411]: now + MonthEnd(2)Out[411]: datetime.datetime(2011, 12, 31, 0, 0)

通过锚点偏移量的rollforward和rollback方法，可显式地将日期向前或向后“滚动”：

In [412]: offset = MonthEnd()In [413]: offset.rollforward(now)Out[413]: datetime.datetime(2011, 11, 30, 0, 0)In [414]: offset.rollback(now)Out[414]: datetime.datetime(2011, 10, 31, 0, 0)

日期偏移量还有一个巧妙的用法，即结合groupby使用这两个“滚动”方法：

In [415]: ts = Series(np.random.randn(20), ...: index=pd.date_range('1/15/2000', periods=20, freq='4d'))In [416]: ts.groupby(offset.rollforward).mean()Out[416]:2000-01-31 -0.4488742000-02-29 -0.6836632000-03-31 0.251920

当然，更简单、更快速地实现该功能的办法是使用resample（稍后将对此进行详细介绍）：

In [417]: ts.resample('M', how='mean')Out[417]:2000-01-31 -0.4488742000-02-29 -0.6836632000-03-31 0.251920Freq: M

译注5

：拿本例来说，就是第一次位移的量可能没有一个月那么长，就在当月。

时区处理

时间序列处理工作中最让人不爽的就是对时区的处理。尤其是夏令时（DST）转变，这是一种最常见的麻烦事。就这一点来说，许多人都选择以协调世界时（UTC，它是格林尼治标准时间（Greenwich Mean Time）的接替者，目前已经是国际标准了）来处理时间序列。时区

是以UTC偏移量的形式表示的。例如，夏令时期间，纽约比UTC慢4小时，而在全年其他时间则比UTC慢5小时。

在Python中，时区信息来自第三方库pytz，它使Python可以使用Olson数据库译注6（汇编了世界时区信息）。这对历史数据非常重要，这是因为由于各地政府的各种突发奇想，夏令时转变日期（甚至UTC偏移量）已经发生过多次改变了。就拿美国来说，DST转变时间自1900年以来就改变过多次！

有关pytz库的更多信息，请查阅其文档。就本书而言，由于pandas包装了pytz的功能，因此你可以不用记忆其API，只要记得时区的名称即可。时区名可以在文档中找到，也可以通过交互的方式查看：

In [418]: import pytzIn [419]: pytz.common_timezones[-5:]Out[419]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']

要从pytz中获取时区对象，使用pytz.timezone即可：

In [420]: tz = pytz.timezone('US/Eastern')In [421]: tzOut[421]:

pandas中的方法既可以接受时区名也可以接受这种对象。我建议只用时区名。

本地化和转换

默认情况下，pandas中的时间序列是单纯的（naive）时区。看看下面这个时间序列：

rng = pd.date_range('3/9/2012 9:30', periods=6, freq='D')ts = Series(np.random.randn(len(rng)), index=rng)

其索引的tz字段为None：

In [423]: print(ts.index.tz)None

在生成日期范围的时候还可以加上一个时区集：

In [424]: pd.date_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC')Out[424]:[2012-03-09 09:30:00, ..., 2012-03-18 09:30:00]Length: 10, Freq: D, Timezone: UTC

从单纯到本地化的转换是通过tz_localize方法处理的：

In [425]: ts_utc = ts.tz_localize('UTC')In [426]: ts_utcOut[426]:2012-03-09 09:30:00+00:00 0.4146152012-03-10 09:30:00+00:00 0.4271852012-03-11 09:30:00+00:00 1.1725572012-03-12 09:30:00+00:00 -0.3515722012-03-13 09:30:00+00:00 1.4545932012-03-14 09:30:00+00:00 2.043319Freq: DIn [427]: ts_utc.indexOut[427]:[2012-03-09 09:30:00, ..., 2012-03-14 09:30:00]Length: 6, Freq: D, Timezone: UTC

一旦时间序列被本地化到某个特定时区，就可以用tz_convert将其转换到别的时区了：

In [428]: ts_utc.tz_convert('US/Eastern')Out[428]:2012-03-09 04:30:00-05:00 0.4146152012-03-10 04:30:00-05:00 0.4271852012-03-11 05:30:00-04:00 1.1725572012-03-12 05:30:00-04:00 -0.3515722012-03-13 05:30:00-04:00 1.4545932012-03-14 05:30:00-04:00 2.043319Freq: D

对于上面这种时间序列（它跨越了美国东部时区的夏令时转变期），我们可以将其本地化到EST，然后转换为UTC或柏林时间：

In [429]: ts_eastern = ts.tz_localize('US/Eastern')In [430]: ts_eastern.tz_convert('UTC')Out[430]:2012-03-09 14:30:00+00:00 0.4146152012-03-10 14:30:00+00:00 0.4271852012-03-11 13:30:00+00:00 1.1725572012-03-12 13:30:00+00:00 -0.3515722012-03-13 13:30:00+00:00 1.4545932012-03-14 13:30:00+00:00 2.043319Freq: DIn [431]: ts_eastern.tz_convert('Europe/Berlin')Out[431]:2012-03-09 15:30:00+01:00 0.4146152012-03-10 15:30:00+01:00 0.4271852012-03-11 14:30:00+01:00 1.1725572012-03-12 14:30:00+01:00 -0.3515722012-03-13 14:30:00+01:00 1.4545932012-03-14 14:30:00+01:00 2.043319Freq: D

tz_localize和tz_convert也是DatetimeIndex的实例方法：

In [432]: ts.index.tz_localize('Asia/Shanghai')Out[432]:[2012-03-09 09:30:00, ..., 2012-03-14 09:30:00]Length: 6, Freq: D, Timezone: Asia/Shanghai

警告：对单纯时间戳的本地化操作还会检查夏令时转变期附近容易混淆或不存在的时间。

操作时区意识型Timestamp对象

跟时间序列和日期范围差不多，Timestamp对象也能被从单纯型（naive）本地化为时区意识型（time zone-aware），并从一个时区转换到另一个时区：

In [433]: stamp = pd.Timestamp('2011-03-12 04:00')In [434]: stamp_utc = stamp.tz_localize('utc')In [435]: stamp_utc.tz_convert('US/Eastern')Out[435]:

在创建Timestamp时，还可以传入一个时区信息：

In [436]: stamp_moscow = pd.Timestamp('2011-03-12 04:00', tz='Europe/Moscow')In [437]: stamp_moscowOut[437]:

时区意识型Timestamp对象在内部保存了一个UTC时间戳值（自UNIX纪元（1970年1月1日）算起的纳秒数）。这个UTC值在时区转换过程中是不会发生变化的：

In [438]: stamp_utc.valueOut[438]: 1299902400000000000In [439]: stamp_utc.tz_convert('US/Eastern').valueOut[439]: 1299902400000000000

当使用pandas的DateOffset对象执行时间算术运算时，运算过程会自动关注是否存在夏令时转变期：

夏令时转变前30分钟In [440]: from pandas.tseries.offsets import HourIn [441]: stamp = pd.Timestamp('2012-03-12 01:30', tz='US/Eastern')In [442]: stampOut[442]: In [443]: stamp + Hour()Out[443]: # 夏令时转变前90分钟In [444]: stamp = pd.Timestamp('2012-11-04 00:30', tz='US/Eastern')In [445]: stampOut[445]: In [446]: stamp + 2 * Hour()Out[446]:

不同时区之间的运算

如果两个时间序列的时区不同，在将它们合并到一起时，最终结果就会是UTC。由于时间戳其实是以UTC存储的，所以这是一个很简单的运算，并不需要发生任何转换：

In [447]: rng = pd.date_range('3/7/2012 9:30', periods=10, freq='B')In [448]: ts = Series(np.random.randn(len(rng)), index=rng)In [449]: tsOut[449]:2012-03-07 09:30:00 -1.7493092012-03-08 09:30:00 -0.3872352012-03-09 09:30:00 -0.2080742012-03-12 09:30:00 -1.2219572012-03-13 09:30:00 -0.0674602012-03-14 09:30:00 0.2290052012-03-15 09:30:00 -0.5762342012-03-16 09:30:00 0.8168952012-03-19 09:30:00 -0.7721922012-03-20 09:30:00 -1.333576Freq: BIn [450]: ts1 = ts[:7].tz_localize('Europe/London')In [451]: ts2 = ts1[2:].tz_convert('Europe/Moscow')In [452]: result = ts1 + ts2In [453]: result.indexOut[453]:[2012-03-07 09:30:00, ..., 2012-03-15 09:30:00]Length: 7, Freq: B, Timezone: UTC

译注6：也叫时区信息数据库，以创始人David Olson命名。

时期及其算术运算

时期（period）表示的是时间区间，比如数日、数月、数季、数年等。Period类所表示的就是这种数据类型，其构造函数需要用到一个字符串或整数，以及表10-4中的频率：

In [454]: p = pd.Period(2007, freq='A-DEC')In [455]: pOut[455]: Period('2007', 'A-DEC')

这个Period对象表示的是从2007年1月1日到2007年12月31日之间的整段时间。只需对Period对象加上或减去一个整数即可达到根据其频率进行位移的效果：

In [456]: p + 5 In [457]: p - 2Out[456]: Period('2012', 'A-DEC') Out[457]: Period('2005', 'A-DEC')

如果两个Period对象拥有相同的频率，则它们的差就是它们之间的单位数量：

In [458]: pd.Period('2014', freq='A-DEC') - pOut[458]: 7

period_range函数可用于创建规则的时期范围：

In [459]: rng = pd.period_range('1/1/2000', '6/30/2000', freq='M')In [460]: rngOut[460]:freq: M[2000-01, ..., 2000-06]length: 6

PeriodIndex类保存了一组Period，它可以在任何pandas数据结构中被用作轴索引：

In [461]: Series(np.random.randn(6), index=rng)Out[461]:2000-01 -0.3091192000-02 0.0285582000-03 1.1296052000-04 -0.3741732000-05 -0.0114012000-06 0.272924Freq: M

PeriodIndex类的构造函数还允许直接使用一组字符串：

In [462]: values = ['2001Q3', '2002Q2', '2003Q1']In [463]: index = pd.PeriodIndex(values, freq='Q-DEC')In [464]: indexOut[464]:freq: Q-DEC[2001Q3, ..., 2003Q1]length: 3

时期的频率转换

Period和PeriodIndex对象都可以通过其asfreq方法被转换成别的频率。假设我们有一个年度时期，希望将其转换为当年年初或年末的一个月度时期。该任务非常简单：

In [465]: p = pd.Period('2007', freq='A-DEC')In [466]: p.asfreq('M', how='start') In [467]: p.asfreq('M', how='end')Out[466]: Period('2007-01', 'M') Out[467]: Period('2007-12', 'M')

你可以将Period('2007','A-DEC')看做一个被划分为多个月度时期的时间段中的游标。图10-1对此进行了说明。对于一个不以12月结束的财政年度，月度子时期的归属情况就不一样了：

In [468]: p = pd.Period('2007', freq='A-JUN')In [469]: p.asfreq('M', 'start') In [470]: p.asfreq('M', 'end')Out[469]: Period('2007-06', 'M') Out[470]: Period('2007-06', 'M')

在将高频率转换为低频率时，超时期（superperiod）是由子时期（subperiod）所属的位置决定的。例如，在A-JUN频率中，月份“2007年8月”实际上是属于周期“2008年”的：

In [471]: p = pd.Period('2007-08', 'M')In [472]: p.asfreq('A-JUN')Out[472]: Period('2008', 'A-JUN')

PeriodIndex或TimeSeries的频率转换方式也是如此：

In [473]: rng = pd.period_range('2006', '2009', freq='A-DEC')In [474]: ts = Series(np.random.randn(len(rng)), index=rng)In [475]: tsOut[475]:2006 -0.6015442007 0.5742652008 -0.1941152009 0.202225Freq: A-DECIn [476]: ts.asfreq('M', how='start') In [477]: ts.asfreq('B', how='end')Out[476]: Out[477]:2006-01 -0.601544 2006-12-29 -0.6015442007-01 0.574265 2007-12-31 0.5742652008-01 -0.194115 2008-12-31 -0.1941152009-01 0.202225 2009-12-31 0.202225Freq: M Freq: B

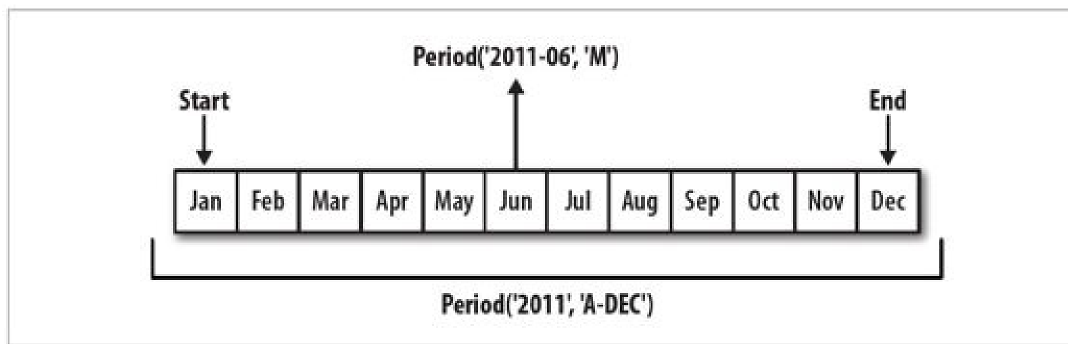


图10-1：Period频率转换示例

按季度计算的时期频率

季度型数据在会计、金融等领域中很常见。许多季度型数据都会涉及“财年末”的概念，通常是一年12个月中某月的最后一个日历日或工作日。就这一点来说，时期"2012Q4"根据财年末的不同会有不同的含义。pandas支持12种可能的季度型频率，即Q-JAN到Q-DEC：

In [478]: p = pd.Period('2012Q4', freq='Q-JAN')In [479]: pOut[479]: Period('2012Q4', 'Q-JAN')

在以1月结束的财年中，2012Q4是从11月到1月（将其转换为日型频率就明白了）。图10-2对此进行了说明：

In [480]: p.asfreq('D', 'start') In [481]: p.asfreq('D', 'end')Out[480]: Period('2011-11-01', 'D') Out[481]: Period('2012-01-31', 'D')

因此，Period之间的算术运算会非常简单。例如，要获取该季度倒数第二个工作日下午4点的时间戳，你可以这样：

In [482]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60In [483]: p4pmOut[483]: Period('2012-01-30 16:00', 'T')In [484]: p4pm.to_timestamp()Out[484]:

Year 2012
M JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC
Q-DEC 2012Q1 2012Q2 2012Q3 2012Q4
Q-SEP 2012Q2 2012Q3 2012Q4 2013Q1
Q-FEB 2012Q4 2013Q1 2013Q2 2013Q3 Q4

图10-2：不同季度型频率之间的转换

period_range还可用于生成季度型范围。季度型范围的算术运算也跟上面是一样的：

In [485]: rng = pd.period_range('2011Q3', '2012Q4', freq='Q-JAN')In [486]: ts = Series(np.arange(len(rng)), index=rng)In [487]: tsOut[487]:2011Q3 02011Q4 12012Q1 22012Q2 32012Q3 42012Q4 5Freq: Q-JANIn [488]: new_rng = (rng.asfreq('B', 'e') -

1).asfreq('T', 's') + 16 * 60In [489]: ts.index = new_rng.to_timestamp()In [490]: tsOut[490]:2010-10-28 16:00:00 02011-01-28 16:00:00 12011-04-28 16:00:00 22011-07-28 16:00:00 32011-10-28 16:00:00 42012-01-30 16:00:00 5

将Timestamp转换为Period（及其反向过程）

通过使用to_period方法，可以将由时间戳索引的Series和DataFrame对象转换为以时期索引：

In [491]: rng = pd.date_range('1/1/2000', periods=3, freq='M')In [492]: ts = Series(randn(3), index=rng)In [493]: pts = ts.to_period()In [494]: tsOut[494]:2000-01-31 -0.5051242000-02-29 2.9544392000-03-31 -2.630247Freq: MIn [495]: ptsOut[495]:2000-01 -0.5051242000-02 2.9544392000-03 -2.630247Freq: M

由于时期指的是非重叠时间区间，因此对于给定的频率，一个时间戳只能属于一个时期。新PeriodIndex的频率默认是从时间戳推断而来的，你也可以指定任何别的频率。结果中允许存在重复时期：

In [496]: rng = pd.date_range('1/29/2000', periods=6, freq='D')In [497]: ts2 = Series(randn(6), index=rng)In [498]: ts2.to_period('M')Out[498]:2000-01 -0.3524532000-01 -0.4778082000-01 0.1615942000-02 1.6868332000-02 0.8219652000-02 -0.667406Freq: M

要转换为时间戳，使用to_timestamp即可：

In [499]: pts = ts.to_period()In [500]: ptsOut[500]:2000-01 -0.5051242000-02 2.9544392000-03 -2.630247Freq: MIn [501]: pts.to_timestamp(how='end')Out[501]:2000-01-31 -0.5051242000-02-29 2.9544392000-03-31 -2.630247Freq: M

通过数组创建PeriodIndex

固定频率的数据集通常会将时间信息分开存放在多个列中。例如，在下面这个宏观经济数据集中，年度和季度就分别存放在不同的列中：

In [502]: data = pd.read_csv('ch08/macrodata.csv')In [503]: data.year In [504]: data.quarterOut[503]: Out[504]:0 1959 0 11 1959 1 22 1959 2 33 1959 3 4... ...199 2008 199 4200 2009 200 1201 2009 201 2202 2009 202 3Name: year, Length: 203 Name: quarter, Length: 203

将这两个数组以及一个频率传入PeriodIndex，就可以将它们合并成DataFrame的一个索引：

In [505]: index = pd.PeriodIndex(year=data.year, quarter=data.quarter, freq='Q-DEC')In [506]: indexOut[506]:freq: Q-DEC[1959Q1, ..., 2009Q3]length: 203In [507]: data.index = indexIn [508]: data.inflOut[508]:1959Q1 0.001959Q2 2.341959Q3 2.741959Q4 0.27...2008Q4 -8.792009Q1 0.942009Q2 3.372009Q3 3.56Freq: Q-DEC, Name: infl, Length: 203

重采样及频率转换

重采样（resampling）指的是将时间序列从一个频率转换到另一个频率的处理过程。将高频率数据聚合到低频率称为降采样（downsampling），而将低频率数据转换到高频率则称为升采样（upsampling）。并不是所有的重采样都能被划分到这两个大类中。例如，将W-WED（每周三）转换为W-FRI既不是降采样也不是升采样。

pandas对象都带有一个resample方法，它是各种频率转换工作的主力函数：

In [509]: rng = pd.date_range('1/1/2000', periods=100, freq='D')In [510]: ts = Series(randn(len(rng)), index=rng)In [511]: ts.resample('M', how='mean')Out[511]:2000-01-31 0.1708762000-02-29 0.1650202000-03-31 0.0954512000-04-30 0.363566Freq: MIn [512]: ts.resample('M', how='mean', kind='period')Out[512]:2000-01 0.1708762000-02 0.1650202000-03 0.0954512000-04 0.363566Freq: M

resample是一个灵活高效的方法，可用于处理非常大的时间序列。我将通过一系列的示例说明其用法。

表10-5：resample方法的参数

参数	说明
freq	表示重采样频率的字符串或DateOffset，例如'M'、'5min'或Second(15)
how='mean'	用于产生聚合值的函数名或数组函数，例如'mean'、'ohlc'、np.max等。默认为'mean'。其他常用的值有：'first'、'last'、'median'、'ohlc'、'max'、'min'
axis=0	重采样的轴，默认为axis=0
fill_method=None	升采样时如何插值，比如'ffill'或'bfill'。默认不插值
closed='right'	在降采样中，各时间段的哪一端是闭合（即包含）的，'right'或'left'。默认为'right'
label='right'	在降采样中，如何设置聚合值的标签，'right'或'left'（面元的右边界或左边界）。例如，9:30到9:35之间的这5分钟会被标记为9:30或9:35。默认为'right'（本例中就是9:35）

表10-5：resample方法的参数（续）

参数	说明
loffset=None	面元标签的时间校正值，比如'-1s' / Second(-1)用于将聚合标签调早1秒
limit=None	在前向或后向填充时，允许填充的最大时期数
kind=None	聚合到时期（'period'）或时间戳（'timestamp'），默认聚合到时间序列的索引类型
convention=None	当重采样时期时，将低频率转换到高频率所采用的约定（'start'或'end'）。默认为'end'

降采样

将数据聚合到规整的低频率是一件非常普通的时间序列处理任务。待聚合的数据不必拥有固定的频率，期望的频率会自动定义聚合的面元边界，这些面元用于将时间序列拆分为多个片段。例如，要转换到月度频率（'M'或'BM'），数据需要被划分到多个单月时间段中。各时间段都是半开放的。一个数据点只能属于一个时间段，所有时间段的并集必须能组成整个时间帧。在用resample对数据进行降采样时，需要考虑两样东西：

·各区间哪边是闭合的。

·如何标记各个聚合面元，用区间的开头还是末尾。

首先，我们来看一些“1分钟”数据：

In [513]: rng = pd.date_range('1/1/2000', periods=12, freq='T')In [514]: ts = Series(np.arange(12), index=rng)In [515]: tsOut[515]:2000-01-01 00:00:00 02000-01-01 00:01:00 12000-01-01 00:02:00 22000-01-01 00:03:00 32000-01-01 00:04:00 42000-01-01 00:05:00 52000-01-01 00:06:00 62000-01-01 00:07:00 72000-01-01 00:08:00 82000-01-01 00:09:00 92000-01-01 00:10:00 102000-01-01 00:11:00 11Freq: T

假设你想要通过求和的方式将这些数据聚合到“5分钟”块中：

In [516]: ts.resample('5min', how='sum')Out[516]:2000-01-01 00:00:00 02000-01-01 00:05:00 152000-01-01 00:10:00 402000-01-01 00:15:00 11Freq: 5T

传入的频率将会以“5分钟”的增量定义面元边界。默认情况下，面元的右边界是包含的，因此00:00到00:05的区间中是包含00:05的

注1

。传入closed='left'会让区间以左边界闭合：

In [517]: ts.resample('5min', how='sum', closed='left')Out[517]:2000-01-01 00:05:00 102000-01-01 00:10:00 352000-01-01 00:15:00 21Freq: 5T

如你所见，最终的时间序列是以各面元右边界的时间戳进行标记的。传入label='left'即可用面元的左边界对其进行标记：

In [518]: ts.resample('5min', how='sum', closed='left', label='left')Out[518]:2000-01-01 00:00:00 102000-01-01 00:05:00 352000-01-01 00:10:00 21Freq: 5T

图10-3说明了“1分钟”数据被转换为“5分钟”数据的处理过程。

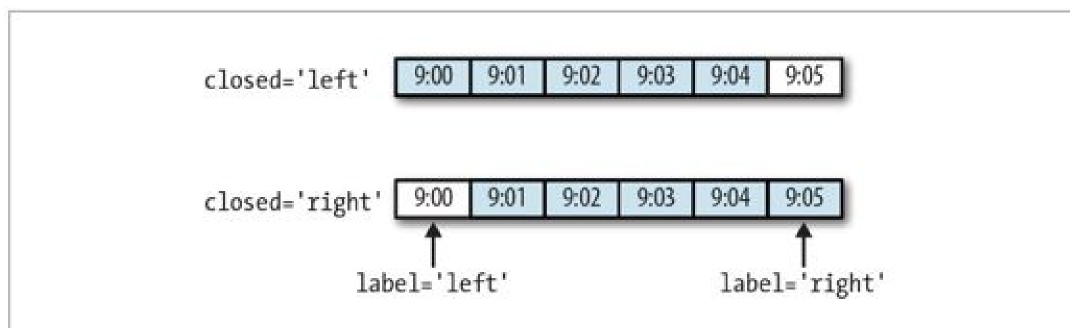


图10-3：各种closed、label约定的“5分钟”重采样演示

最后，你可能希望对结果索引做一些位移，比如从右边界减去一秒以便更容易明白该时间戳到底表示的是哪个区间。只需通过loffset设置一个字符串或日期偏移量即可实现这个目的：

In [519]: ts.resample('5min', how='sum', loffset='-1s')Out[519]:1999-12-31 23:59:59 02000-01-01 00:04:59 152000-01-01 00:09:59 402000-01-01 00:14:59 11Freq: 5T

此外，也可以通过调用结果对象的shift方法来实现该目的，这样就不需要设置loffset了。

OHLC重采样

金融领域中有一种无所不在的时间序列聚合方式，即计算各面元的四个值：第一个值（open，开盘）、最后一个值（close，收盘）、最大值（high，最高）以及最小值（low，最低）。传入how='ohlc'即可得到一个含有这四种聚合值的DataFrame。整个过程很高效，只需一次扫描即可计算出结果：

In [520]: ts.resample('5min', how='ohlc')Out[520]: open high low close2000-01-01 00:00:00 0 0 0 02000-01-01 00:05:00 1 5 1 52000-01-01 00:10:00 6 10 6 102000-01-01 00:15:00 11 11 11 11

通过groupby进行重采样

另一种降采样的办法是使用pandas的groupby功能。例如，你打算根据月份或星期几进行分组，只需传入一个能够访问时间序列的索引上的这些字段的函数即可：

In [521]: rng = pd.date_range('1/1/2000', periods=100, freq='D')In [522]: ts = Series(np.arange(100), index=rng)In [523]: ts.groupby(lambda x: x.month).mean()Out[523]:1 152 453 754 95In [524]: ts.groupby(lambda x: x.weekday).mean()Out[524]:0 47.51 48.52 49.53 50.54 51.55 49.06 50.0

升采样和插值

在将数据从低频率转换到高频率时，就不需要聚合了。我们来看一个带有一些周型数据的DataFrame：

In [525]: frame = DataFrame(np.random.randn(2, 4), ...: index=pd.date_range('1/1/2000', periods=2, freq='W-WED'), ...: columns=['Colorado', 'Texas', 'New York', 'Ohio'])In [526]: frame[:5]Out[526]: Colorado Texas New York Ohio2000-01-05 -0.609657 -0.268837

0.195592 0.859792000-01-12 -0.263206 1.141350 -0.101937 -0.07666

将其重采样到日频率，默认会引入缺失值：

In [527]: df_daily = frame.resample('D')In [528]: df_dailyOut[528]: Colorado Texas New York Ohio2000-01-05 -0.609657 -0.268837 0.195592 0.859792000-01-06 NaN NaN NaN NaN2000-01-07 NaN NaN NaN NaN2000-01-08 NaN NaN NaN NaN2000-01-09 NaN NaN NaN NaN2000-01-10 NaN NaN NaN NaN2000-01-11 NaN NaN NaN NaN2000-01-12 -0.263206 1.141350 -0.101937 -0.07666

假设你想要用前面的周型值填充“非星期三”。resampling的填充和插值方式跟fillna和reindex的一样：

In [529]: frame.resample('D', fill_method='ffill')Out[529]: Colorado Texas New York Ohio2000-01-05 -0.609657 -0.268837 0.195592 0.859792000-01-06 -0.609657 -0.268837 0.195592 0.859792000-01-07 -0.609657 -0.268837 0.195592 0.859792000-01-08 -0.609657 -0.268837 0.195592 0.859792000-01-09 -0.609657 -0.268837 0.195592 0.859792000-01-10 -0.609657 -0.268837 0.195592 0.859792000-01-11 -0.609657 -0.268837 0.195592 0.859792000-01-12 -0.263206 1.141350 -0.101937 -0.07666

同样，这里也可以只填充指定的时期数（目的是限制前面的观测值的持续使用距离）：

In [530]: frame.resample('D', fill_method='ffill', limit=2)Out[530]: Colorado Texas New York Ohio2000-01-05 -0.609657 -0.268837 0.195592 0.859792000-01-06 -0.609657 -0.268837 0.195592 0.859792000-01-07 -0.609657 -0.268837 0.195592 0.859792000-01-08 NaN NaN NaN NaN2000-01-09 NaN NaN NaN NaN2000-01-10 NaN NaN NaN NaN2000-01-11 NaN NaN NaN NaN2000-01-12 -0.263206 1.141350 -0.101937 -0.07666

注意，新的日期索引完全没必要跟旧的相交：

In [531]: frame.resample('W-THU', fill_method='ffill')Out[531]: Colorado Texas New York Ohio2000-01-06 -0.609657 -0.268837 0.195592 0.859792000-01-13 -0.263206 1.141350 -0.101937 -0.07666

通过时期进行重采样

对那些使用时期索引的数据进行重采样是件非常简单的事情：

In [532]: frame = DataFrame(np.random.randn(24, 4), ...: index=pd.period_range('1-2000', '12-2001', freq='M'), ...: columns=['Colorado', 'Texas', 'New York', 'Ohio'])In [533]: frame[:5]Out[533]: Colorado Texas New York Ohio2000-01 0.120837 1.076607 0.434200 0.0564322000-02 -0.378890 0.047831 0.341626 1.5679202000-03 -0.047619 -0.821825 -0.179330 -0.1666752000-04 0.333219 -0.544615 -0.653635 -2.3110262000-05 1.612270

-0.806614 0.557884 0.580201In [534]: annual_frame = frame.resample('A-DEC', how='mean')In [535]: annual_frameOut[535]: Colorado Texas New York Ohio2000 0.352070 -0.553642 0.196642 -0.0940992001 0.158207 0.042967 -0.360755 0.184687

升采样要稍微麻烦一些，因为你必须决定在新频率中各区间的哪端用于放置原来的值，就像asfreq方法那样。convention参数默认为'end'，可设置为'start'：

Q-DEC: 季度型（每年以12月结束）In [536]: annual_frame.resample('Q-DEC', fill_method='ffill')Out[536]: Colorado Texas New York Ohio2000Q4 0.352070 -0.553642 0.196642 -0.0940992001Q1 0.352070 -0.553642 0.196642 -0.0940992001Q2 0.352070 -0.553642 0.196642 -0.0940992001Q3 0.352070 -0.553642 0.196642 -0.0940992001Q4 0.158207 0.042967 -0.360755 0.184687In [537]: annual_frame.resample('Q-DEC', fill_method='ffill', convention='start')Out[537]: Colorado Texas New York Ohio2000Q1 0.352070 -0.553642 0.196642 -0.0940992000Q2 0.352070 -0.553642 0.196642 -0.0940992000Q3 0.352070 -0.553642 0.196642 -0.0940992000Q4 0.352070 -0.553642 0.196642 -0.0940992001Q1 0.158207 0.042967 -0.360755 0.184687

由于时期指的是时间区间，所以升采样和降采样的规则就比较严格：

·在降采样中，目标频率必须是源频率的子时期（subperiod）。

·在升采样中，目标频率必须是源频率的超时期（superperiod）。

如果不满足这些条件，就会引发异常。这主要影响的是按季、年、周计算的频率。例如，由Q-MAR定义的时间区间只能升采样为A-MAR、A-JUN、A-SEP、A-DEC等：

In [538]: annual_frame.resample('Q-MAR', fill_method='ffill')Out[538]: Colorado Texas New York Ohio2001Q3 0.352070 -0.553642 0.196642 -0.0940992001Q4 0.352070 -0.553642 0.196642 -0.0940992002Q1 0.352070 -0.553642 0.196642 -0.0940992002Q2 0.352070 -0.553642 0.196642 -0.0940992002Q3 0.158207 0.042967 -0.360755 0.184687

注1

：closed='right'、label='right'这两个默认值可能会让部分用户感到奇怪。在实际工作当中，这两个选项的值比较随意。对于某些目标频率，c l o s e d='l e f t'会更好，而对于其他的，则closed='right'才更为合理。你真正应该关注的是要如何对数据分段。

时间序列绘图

pandas时间序列的绘图功能在日期格式化方面比matplotlib原生的要好。来看下面这个例子，我先从Yahoo!Finance下载了几只美国股票的一些价格数据：

In [539]: close_px_all = pd.read_csv('ch09/stock_px.csv', parse_dates=True, index_col=0)In [540]: close_px = close_px_all[['AAPL', 'MSFT', 'XOM']]In [541]: close_px = close_px.resample('B', fill_method='ffill')In [542]: close_pxOut[542]:DatetimeIndex: 2292 entries, 2003-01-02 00:00:00 to 2011-10-14 00:00:00Freq: BData columns:AAPL 2292 non-null valuesMSFT 2292 non-null valuesXOM 2292 non-null valuesdtypes: float64(3)

对其中任意一列调用plot即可生成一张简单的图表，如图10-4所示。

In [544]: close_px['AAPL'].plot()

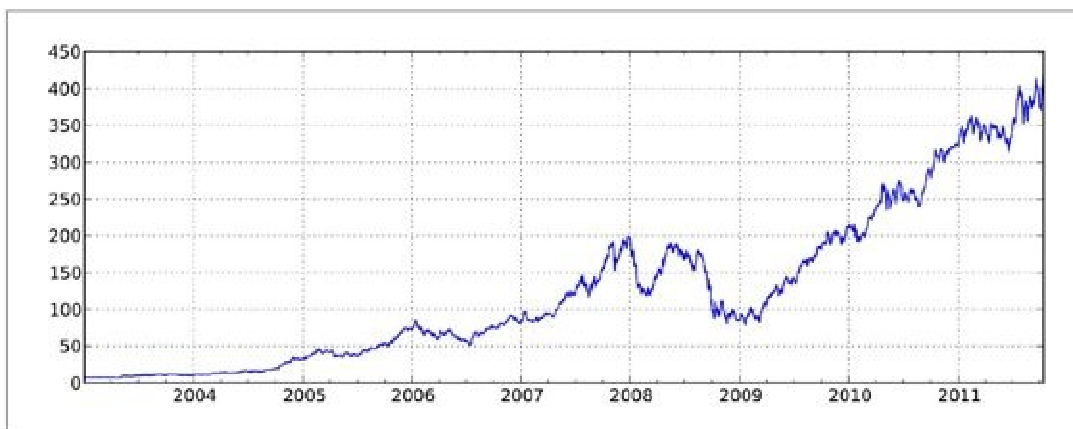


图10-4：AAPL每日价格

当对DataFrame调用plot时，所有时间序列都会被绘制在一个subplot上，并有一个图例说明哪个是哪个。这里我只绘制了2009年的数据，如图10-5所示，月份和年度都被格式化到了X轴上。

In [546]: close_px.ix['2009'].plot()

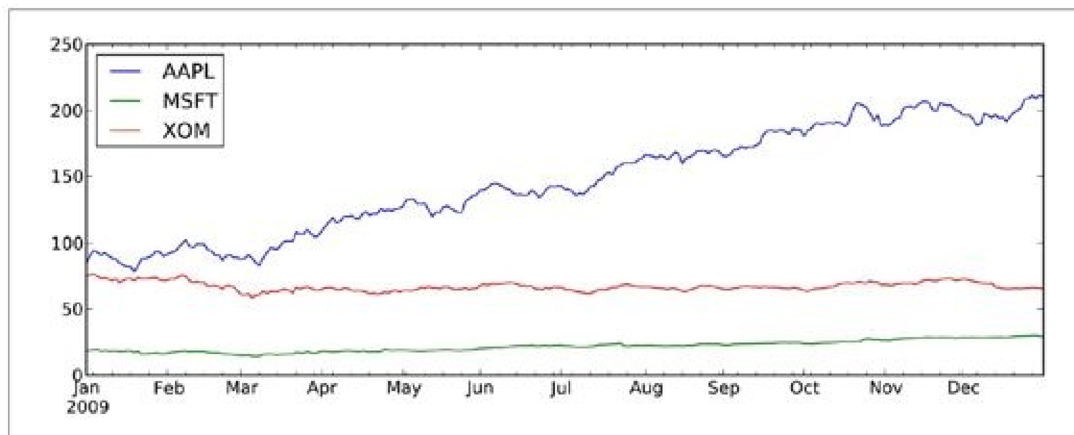


图10-5：2009年的股票价格

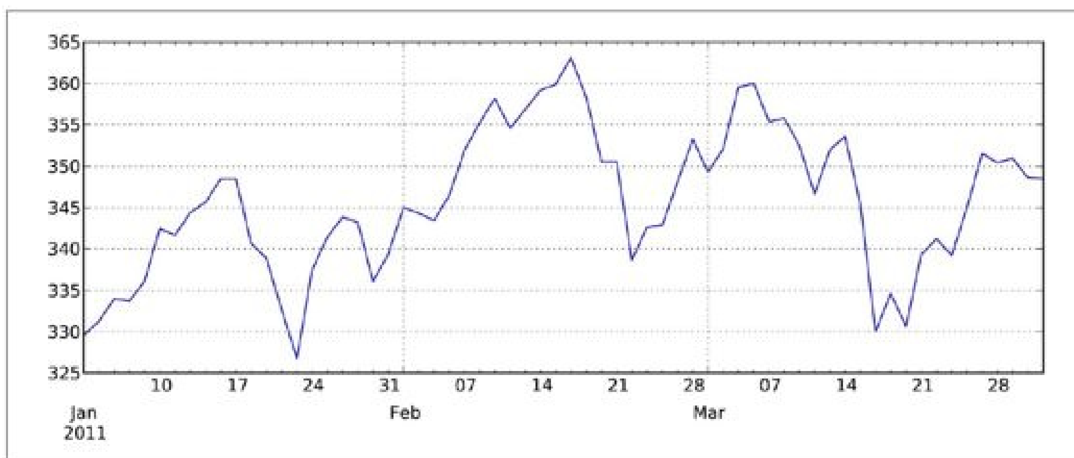


图10-6：苹果公司在2011年1月到3月间的每日股价

图10-6展示了苹果公司在2011年1月到3月间的每日股价。

In [548]: close_px['AAPL'].ix['01-2011':'03-2011'].plot()

季度型频率的数据会用季度标记进行格式化，这种事情如果纯手工做的话那是很费精力的。如图10-7所示。

In [550]: appl_q = close_px['AAPL'].resample('Q-DEC', fill_method='ffill')In [551]: appl_q.ix['2009':].plot()

pandas时间序列在绘图方面还有一个特点：当右键点击

译注7

并拖拉（放大、缩小）时，日期会动态展开或收缩，且绘图窗口中的时间区间会被重新格式化。当然，只有在交互模式下使用matplotlib才会有此效果。

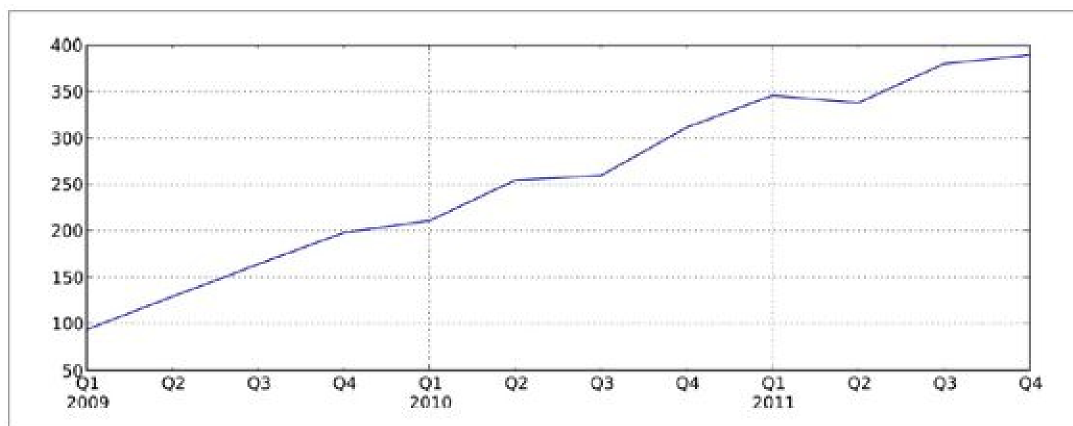


图10-7：苹果公司在2009年到2011年间的每季度股价

译注7：应该是按住（hold）而不是点击（click）。

移动窗口函数

在移动窗口（可以带有指数衰减权数）上计算的各种统计函数也是一类常见于时间序列的数组变换。我将它们称为移动窗口函数（moving window function），其中还包括那些窗口不定长的数（如指数加权移动平均）。跟其他统计函数一样，移动窗口函数也会自动排除缺失值。

rolling_mean是其中最简单的一个。它接受一个TimeSeries或DataFrame以及一个window（表示期数）：

In [555]: close_px.AAPL.plot()Out[555]: In [556]: pd.rolling_mean(close_px.AAPL, 250).plot()

结果如图10-8所示。默认情况下，诸如rolling_mean这样的函数需要指定数量

译注8

的非NA观测值。可以修改该行为以解决缺失数据的问题。其实，在时间序列开始处尚不足窗口期的那些数据就是个特例（见图10-9）：

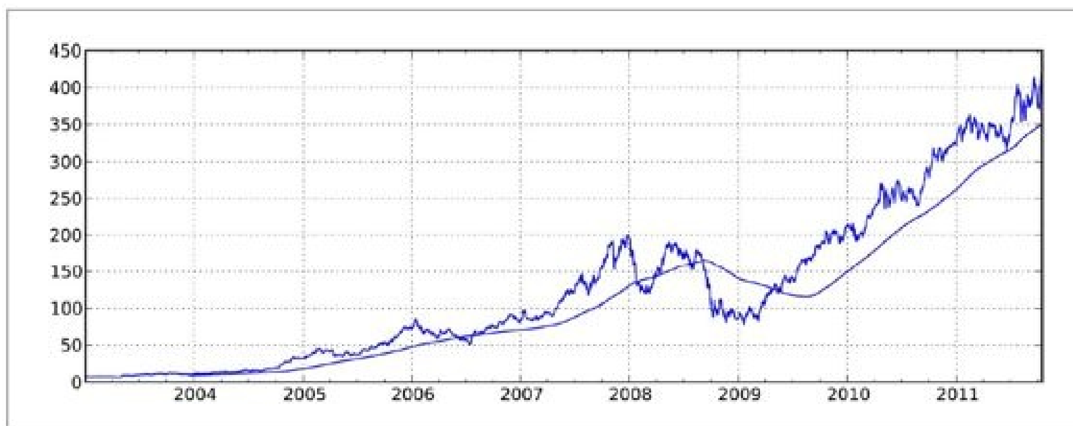


图10-8：苹果公司股价的250日均线

In [558]: appl_std250 = pd.rolling_std(close_px.AAPL, 250, min_periods=10)In [559]: appl_std250[5:12]Out[559]:2003-01-09 NaN2003-01-10 NaN2003-01-13 NaN2003-01-14 NaN2003-01-15 0.0774962003-01-16 0.0747602003-01-17 0.112368Freq: BIn [560]: appl_std250.plot()

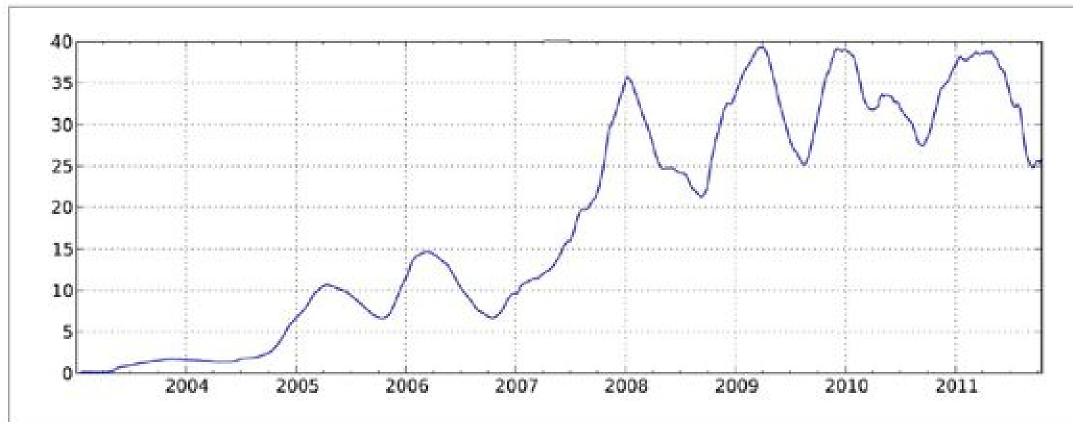


图10-9：苹果公司250日每日回报标准差

要计算扩展窗口平均（expanding window mean），你可以将扩展窗口看做一个特殊的窗口，其长度与时间序列一样，但只需一期（或多期

译注9

）即可计算一个值：

通过rolling_mean定义扩展平均In [561]: expanding_mean = lambda x: rolling_mean(x, len(x), min_periods=1)

对DataFrame调用rolling_mean（以及与之类似的函数）会将转换应用到所有的列上（见图10-10）：

In [563]: pd.rolling_mean(close_px, 60).plot(logy=True)

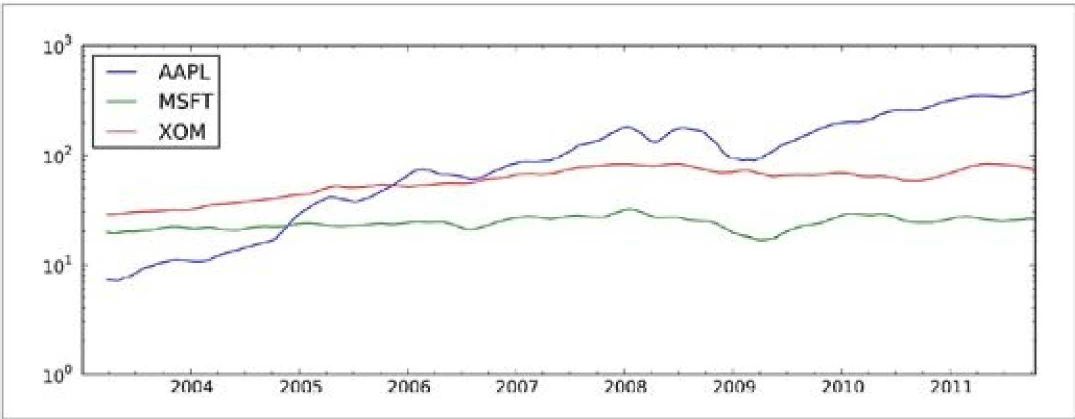


图10-10：各股价60日均线（对数Y轴）

表10-6中列出了pandas中的此类函数。

表10-6：移动窗口和指数加权函数

函数	说明
rolling_count	返回各窗口非NA观测值的数量
rolling_sum	移动窗口的和
rolling_mean	移动窗口的平均值
rolling_median	移动窗口的中位数
rolling_var、rolling_std	移动窗口的方差和标准差。分母为n－1
rolling_skew、olling_kurt	移动窗口的偏度（三阶矩）和峰度（四阶矩）
rolling_min、rolling_max	移动窗口的最小值和最大值
rolling_quantile	移动窗口指定百分位数/样本分位数位置的值
rolling_corr、rolling_cov	移动窗口的相关系数和协方差
rolling_apply	对移动窗口应用普通数组函数
ewma	指数加权移动平均
ewmvar、ewmstd	指数加权移动方差和标准差
ewmcorr、ewmcov	指数加权移动相关系数和协方差

注意： bottleneck（由Keith Goodman制作的Python库）提供了另一种对NaN友好的移动窗口函数集。值得一看，说不定能在你的工作中派上用场。

指数加权函数

另一种使用固定大小窗口及相等权数观测值的办法是，定义一个衰减因子（decay factor）常量，以便使近期的观测值拥有更大的权数。用数学术语来讲，如果mat是时间t的移动平均结果，x是时间序列，结果中的各个值可用mat =*amat-1+(a -1)*x-t进行计算，其中a为衰减因子。衰减因子的定义方式有很多，比较流行的是使用时间间隔（span），它可以使结果兼容于窗口大小等于时间间隔的简单移动窗口（simple moving window）函数。

由于指数加权统计会赋予近期的观测值更大的权数，因此相对于等权统计

译注10

，它能“适应”更快的变化。下面这个例子对比了苹果公司股价的60日移动平均和span=60的指数加权移动平均（如图10-11所示）：

fig, axes = plt.subplots(nrows=2, ncols=1, sharex=True, sharey=True, figsize=(12, 7))aapl_px = close_px.AAPL['2005':'2009']ma60 = pd.rolling_mean(aapl_px, 60, min_periods=50)ewma60 = pd.ewma(aapl_px, span=60)aapl_px.plot(style='k-', ax=axes[0])ma60.plot(style='k--', ax=axes[0])aapl_px.plot(style='k-', ax=axes[1])ewma60.plot(style='k--', ax=axes[1])axes[0].set_title('Simple MA')axes[1].set_title('Exponentially-weighted MA')

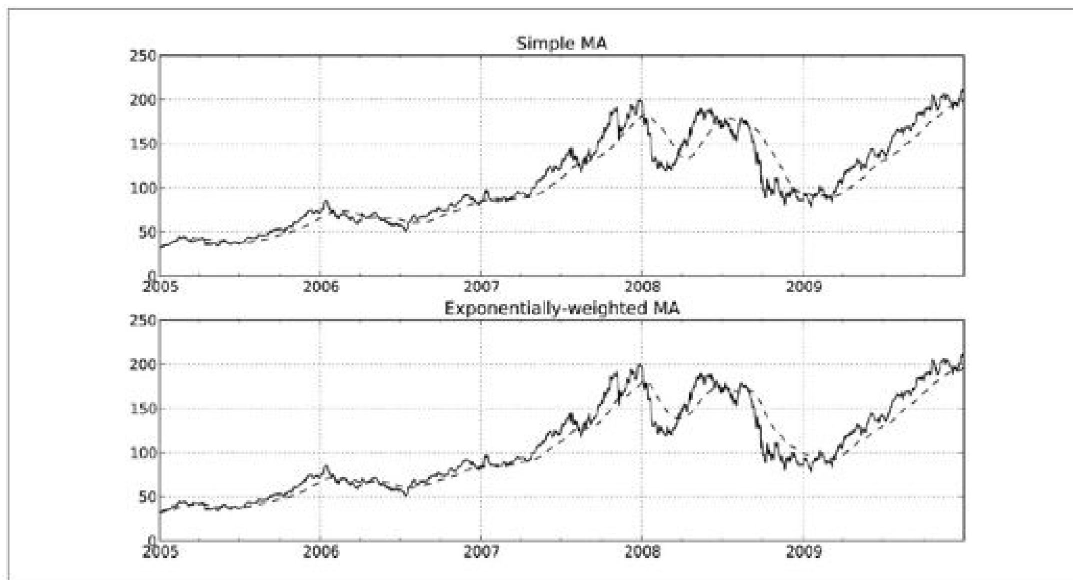


图10-11：简单移动平均与指数加权移动平均

二元移动窗口函数

有些统计运算（如相关系数和协方差）需要在两个时间序列上执行。例如，金融分析师常常对某只股票对某个参考指数（如标准普尔500指数）的相关系数感兴趣。我们可以通过计算百分数变化并使用rolling_corr的方式得到该结果（如图10-12所示）：

In [569]: spx_px = close_px_all ['SPX']In [570]: spx_rets = spx_px / spx_px.shift(1) - 1In [571]: returns = close_px.pct_change()In [572]: corr = pd.rolling_corr(returns.AAPL, spx_rets, 125, min_periods=100)In [573]: corr.plot()

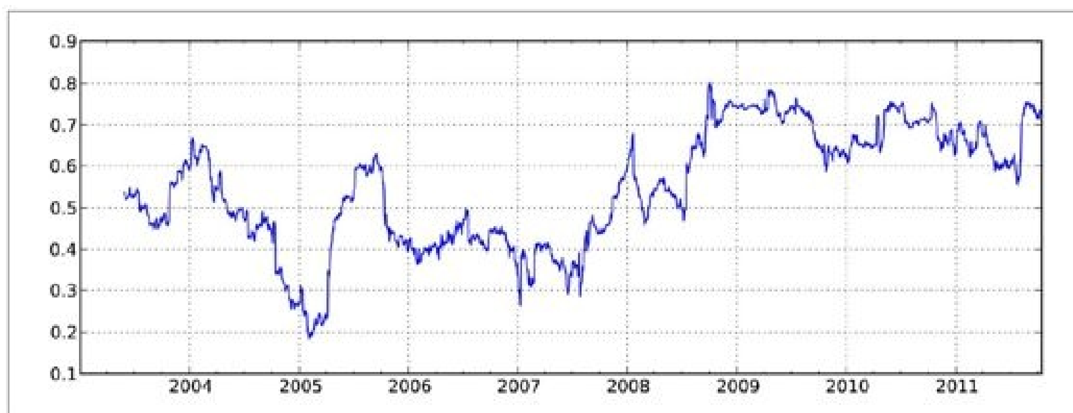


图10-12：AAPL 6个月的回报与标准普尔500指数的相关系数

假设你想要一次性计算多只股票与标准普尔500指数的相关系数。虽然编写一个循环并新建一个DataFrame不是什么难事，但比较啰。其实，只需传入一个TimeSeries和一个DataFrame，rolling_corr就会自动计算TimeSeries（本例中就是spx_rets）与DataFrame各列的相关系数。结果如图10-13所示：

In [575]: corr = pd.rolling_corr(returns, spx_rets, 125, min_periods=100)In [576]: corr.plot()

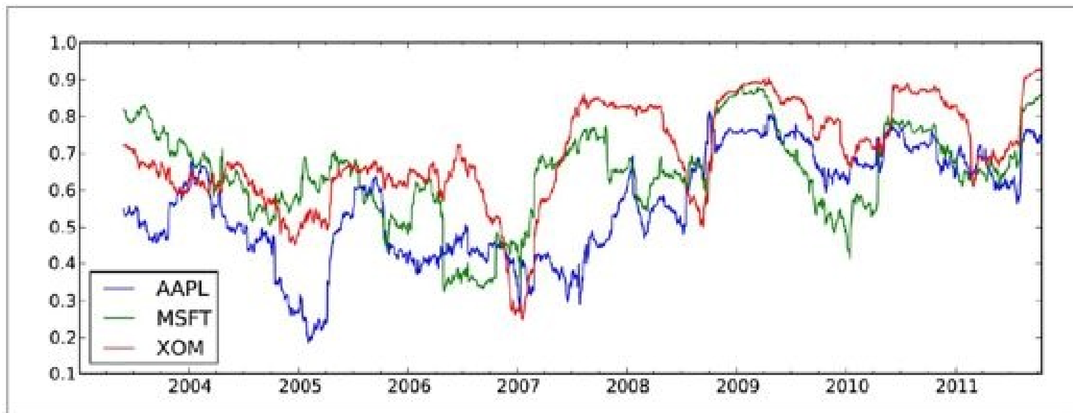


图10-13：3只股票6个月的回报与标准普尔500指数的相关系数

用户定义的移动窗口函数

rolling_apply函数使你能够在移动窗口上应用自己设计的数组函数。唯一要求的就是：该函数要能从数组的各个片段中产生单个值（即约简）。比如说，当我们用rolling_quantile计算样本分位数时，可能对样本中特定值的百分等级感兴趣。scipy.stats.percentileofscore函数就能达到这个目的：

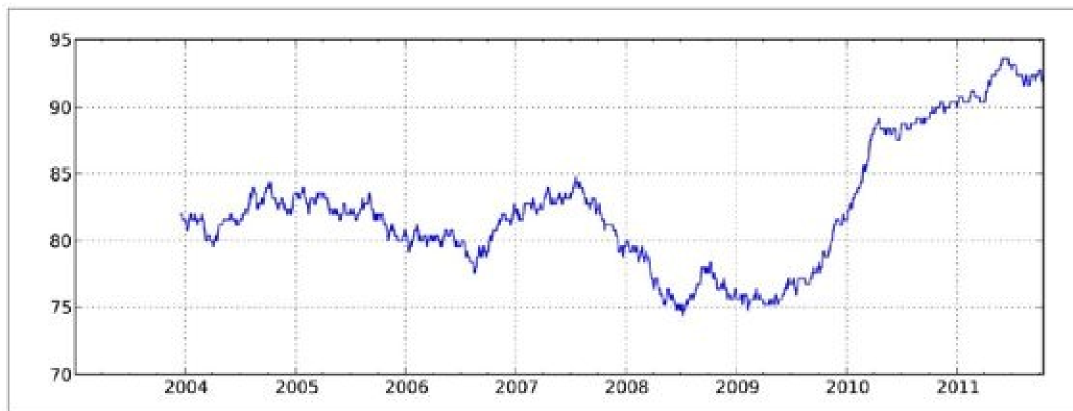


图10-14：AAPL 2%回报率的百分等级（一年窗口期）

In [578]: from scipy.stats import percentileofscoreIn [579]: score_at_2percent = lambda x: percentileofscore(x, 0.02)In [580]: result = pd.rolling_apply(returns.AAPL, 250, score_at_2percent)In [581]: result.plot()

译注8：这是针对窗口而言的，即一个窗口里面必须有多少个非NA值。

译注9：不设置就全空，也不太大，大了就无意义了。

译注10

：就是不加权的普通移动平均。

性能和内存使用方面的注意事项

Timestamp和Period都是以64位整数表示的（即NumPy的datetime64数据类型）。也就是说，对于每个数据点，其时间戳需要占用8字节的内存。因此，含有一百万个float64数据点时间序列需要占用大约16MB的内存空间。由于pandas会尽量在多个时间序列之间共享索引，所以创建现有时间序列的视图不会占用更多内存译注11。此外，低频率索引（日以上）会被存放在一个中心缓存中，所以任何固定频率的索引都是该日期缓存的视图。所以，如果你有一个很大的低频率时间序列，索引所占用的内存空间将不会很大。

性能方面，pandas对数据对齐（两个不同索引的ts1+ts2的幕后工作）和重采样运算进行了高度优化。下面这个例子将一亿个数据点聚合为OHLC：

In [582]: rng = pd.date_range('1/1/2000', periods=10000000, freq='10ms')In [583]: ts = Series(np.random.randn(len(rng)), index=rng)In [584]: tsOut[584]:2000-01-01 00:00:00 -1.4022352000-01-01 00:00:00.010000 2.4246672000-01-01 00:00:00.020000 -1.9560422000-01-01 00:00:00.030000 -0.897339...2000-01-02 03:46:39.960000 0.4955302000-01-02 03:46:39.970000 0.5747662000-01-02 03:46:39.980000 1.3483742000-01-02 03:46:39.990000 0.665034Freq: 10L, Length: 10000000In [585]: ts.resample('15min', how='ohlc')Out[585]:DatetimeIndex: 113 entries, 2000-01-01 00:00:00 to 2000-01-02 04:00:00Freq: 15TData columns:open 113 non-null valueshigh 113 non-null valueslow 113 non-null valuesclose 113 non-null valuesdtypes: float64(4)In [586]: %timeit ts.resample('15min', how='ohlc')10 loops, best of 3: 61.1 ms per loop

运行时间跟聚合结果的相对大小有一定关系，越高频率的聚合所耗费的时间越多：

In [587]: rng = pd.date_range('1/1/2000', periods=10000000, freq='1s')In [588]: ts = Series(np.random.randn(len(rng)), index=rng)In [589]: %timeit ts.resample('15s', how='ohlc')1 loops, best of 3: 88.2 ms per loop

可能在你阅读本书的时候，这些算法的性能已经大为改进了。比如说，目前并没有对规则频率之间的转换做任何优化，但这肯定是要做的。

译注11：原文就是这么表达的。我感到很不解，既然是视图，当然不会占用多少内容，毕竟就是比单个指针大点的东西而已。结合上下文来看，估计说的只是索引的问题。

第11章　金融和经济数据应用

从2005年开始，Python在金融行业中的应用越来越多，这主要得益于众多成熟的函数库（NumPy和pandas）以及大量经验丰富的Python程序员。许多机构都发现Python不仅非常适成为交互式的分析环境，也非常适合开发稳健的系统，而且所需的时间比Java或C++少得

多。Python还是一种非常好的粘合层，可以非常轻松地为C或C++编写的库构建Python接口。

金融分析领域的内容博大精深，甚至拿一整本书来讲都不为过，在这里我只是希望告诉你如何利用本书中的工具去解决金融领域中的一些特殊问题。跟其他的研究和分析领域一样，在数据规整化方面所花费的精力常常会比解决核心建模和研究问题所花费的要多得多。就是因为找不到合适的数据处理工具，所以我才在2008年开始创立pandas的。

在本章的示例中，我将使用术语“截面”（cross-section）来表示某个时间点的数据。例如，标准普尔500指数中所有成分股在特定日期的收盘价就形成了一个截面。多个数据项（例如价格和成交量）在多个时间点的截面数据就构成了一个面板（panel）。面板数据既可以被表示为层次化索引的DataFrame，也可以被表示为三维的Panel pandas对象。

数据规整化方面的话题

前面几章中陆陆续续介绍过一些不错的数据规整化工具。这里，我将着重介绍一些跟金融问题域有关的话题。

时间序列以及截面对齐

在处理金融数据时，最费神的一个问题就是所谓的“数据对齐”（data alignment）问题。两个相关的时间序列的索引可能没有很好的对齐，或两个DataFrame对象可能含有不匹配的列或行。MATLAB、R以及其他矩阵编程语言的用户常常需要花费大量的精力将数据规整化为完全对齐的形式。以我的经验来看，手工处理数据对齐问题是一件令人非常郁闷的工作，而验证数据是否对齐则还要更郁闷一些。不仅如此，合并未对齐的数据还很有可能带来各种bug。

pandas可以在算术运算中自动对齐数据。在实际工作当中，这不仅能为你带来极大的自由度，而且还能提高你的工作效率。来看下面这两个DataFrame，它们分别含有股票价格和成交量的时间序列译注1：

In [16]: pricesOut[16]: AAPL JNJ SPX XOM2011-09-06 379.74 64.64 1165.24 71.152011-09-07 383.93 65.43 1198.62 73.652011-09-08 384.14 64.95 1185.90 72.822011-09-09 377.48 63.64 1154.23 71.012011-09-12 379.94 63.59 1162.27 71.842011-09-13 384.62 63.61 1172.87 71.652011-09-14 389.30 63.73 1188.68 72.64In [17]: volumeOut[17]: AAPL JNJ XOM2011-09-06 18173500 15848300 254163002011-09-07 12492000 10759700 231084002011-09-08 14839800 15551500 224348002011-09-09 20171900 17008200 279691002011-09-12 16697300 13448200 26205800

假设你想要用所有有效数据计算一个成交量加权平均价格（为了简单起见，假设成交量数据是价格数据的子集）。由于pandas会在算术运算过程中自动将数据对齐，并在sum这样的函数中排除缺失数据，所以我们只需编写下面这条简洁的表达式即可：

In [18]: prices *volumeOut[18]: AAPL JNJ SPX XOM2011-09-06 6901204890 1024434112 NaN 18083697452011-09-07 4796053560 704007171 NaN 17019336602011-09-08 5700560772 1010069925 NaN 16337021362011-09-09 7614488812 1082401848 NaN 19860857912011-09-12 6343972162 855171038 NaN 18826246722011-09-13 NaN NaN*

NaN NaN2011-09-14 NaN NaN NaN NaNIn [19]: vwap = (prices volume).sum() / volume.sum()In [20]: vwap In [21]: vwap.dropna()Out[20]: Out[21]:AAPL 380.655181 AAPL 380.655181JNJ 64.394769 JNJ 64.394769SPX NaN XOM 72.024288XOM 72.024288

由于SPX在volume中找不到，所以你随时可以显式地将其丢弃。如果希望手工进行对齐，可以使用DataFrame的align方法，它返回的是一个元组，含有两个对象的重索引版本：

In [22]: prices.align(volume, join='inner')Out[22]:(AAPL JNJ XOM2011-09-06 379.74 64.64 71.152011-09-07 383.93 65.43 73.652011-09-08 384.14 64.95 72.822011-09-09 377.48 63.64 71.012011-09-12 379.94 63.59 71.84, AAPL JNJ XOM2011-09-06 18173500 15848300 254163002011-09-07 12492000 10759700 231084002011-09-08 14839800 15551500 224348002011-09-09 20171900 17008200 279691002011-09-12 16697300 13448200 26205800)

另一个不可或缺的功能是，通过一组索引可能不同的Series构建一个DataFrame。

In [23]: s1 = Series(range(3), index=['a', 'b', 'c'])In [24]: s2 = Series(range(4), index=['d', 'b', 'c', 'e'])In [25]: s3 = Series(range(3), index=['f', 'a', 'c'])In [26]: DataFrame({'one': s1, 'two': s2, 'three': s3})Out[26]: one three twoa 0 1 NaNb 1 NaN 1c 2 2 2d NaN NaN 0e NaN NaN 3f NaN 0 NaN

跟前面一样，这里也可以显式定义结果的索引（丢弃其余的数据）：

In [27]: DataFrame({'one': s1, 'two': s2, 'three': s3}, index=list('face'))Out[27]: one three twof NaN 0 NaNa 0 1 NaNc 2 2 2e NaN NaN 3

频率不同的时间序列的运算

经济学时间序列常常有着按年、季、月、日计算的或其他更特殊的频率。有些完全就是不规则的，比如说，盈利预测调整随时都可能会发生。频率转换和重对齐的两大主要工具是resample和reindex方法。resample用于将数据转换到固定频率，而reindex则用于使数据符合一个新索引。它们都支持插值（如前向填充）逻辑。

来看一个简单的周型时间序列：

In [28]: ts1 = Series(np.random.randn(3), ...: index=pd.date_range('2012-6-13', periods=3, freq='W-WED'))In [29]: ts1Out[29]:2012-06-13 -1.1248012012-06-20 0.4690042012-06-27 -0.117439Freq: W-WED

如果将其重采样到工作日（星期一到星期五）频率，则那些没有数据的日子就会出现一个“空洞”：

In [30]: ts1.resample('B')Out[30]:2012-06-13 -1.1248012012-06-14 NaN2012-06-15 NaN2012-06-18 NaN2012-06-19 NaN2012-06-20 0.4690042012-06-21 NaN2012-06-22 NaN2012-06-25 NaN2012-06-26 NaN2012-06-27 -0.117439Freq: B

当然，只需将fill_method设置为'ffill'即可用前面的值填充这些空白。处理较低频率的数据时常常这么干，因为最终结果中各时间点都有一个最新的有效值：

In [31]: ts1.resample('B', fill_method='ffill')Out[31]:2012-06-13 -1.1248012012-06-14 -1.1248012012-06-15 -1.1248012012-06-18 -1.1248012012-06-19 -1.1248012012-06-20 0.4690042012-06-21 0.4690042012-06-22 0.4690042012-06-25 0.4690042012-06-26 0.4690042012-06-27 -0.117439Freq: B

在实际工作当中，将较低频率的数据升采样到较高的规整频率是一种不错的解决方案，但是对于更一般化的不规整时间序列可能就不太合适了。看看下面这个不规整样本的时间序列（各时间点更一般化）：

In [32]: dates = pd.DatetimeIndex(['2012-6-12', '2012-6-17', '2012-6-18', ...: '2012-6-21', '2012-6-22', '2012-6-29'])In [33]: ts2 = Series(np.random.randn(6), index=dates)In [34]: ts2Out[34]:2012-06-12 -0.4494292012-06-17 0.4596482012-06-18 -0.1725312012-06-21 0.8359382012-06-22 -0.5947792012-06-29 0.027197

如果要将ts1中“最当前”的值（即前向填充）加到ts2上。一个办法是将两者重采样为规整频率后再相加，但是如果想维持ts2中的日期索引，则reindex会是一种更好的解决方案：

In [35]: ts1.reindex(ts2.index, method='ffill')Out[35]:2012-06-12 NaN2012-06-17 -1.1248012012-06-18 -1.1248012012-06-21 0.4690042012-06-22 0.4690042012-06-29 -0.117439In [36]: ts2 + ts1.reindex(ts2.index, method='ffill')Out[36]:2012-06-12 NaN2012-06-17 -0.6651532012-06-18 -1.2973322012-06-21 1.3049422012-06-22 -0.1257752012-06-29 -0.090242

使用Period

Period（表示时间区间）提供了另一种处理不同频率时间序列的办法，尤其是那些有着特殊规范的以年或季度为频率的金融或经济序列。比如说，一个公司可能会发布其以6月结尾的财年的每季度盈利报告，即频率为Q-JUN。来看两个有关GDP和通货膨胀的宏观经济时间序列：

In [37]: gdp = Series([1.78, 1.94, 2.08, 2.01, 2.15, 2.31, 2.46], ...: index=pd.period_range('1984Q2', periods=7, freq='Q-SEP'))In [38]: infl = Series([0.025, 0.045, 0.037, 0.04], ...: index=pd.period_range('1982', periods=4, freq='A-DEC'))In [39]: gdp In [40]: inflOut[39]: Out[40]:1984Q2 1.78 1982 0.0251984Q3 1.94 1983 0.0451984Q4 2.08 1984 0.0371985Q1 2.01 1985 0.0401985Q2 2.15 Freq: A-DEC1985Q3 2.311985Q4 2.46Freq: Q-SEP

跟Timestamp的时间序列不同，由Period索引的两个不同频率的时间序列之间的运算必须进行显式转换。在本例中，假设已知infl值是在每年年末观测的，于是我们就可以将其转换到Q-SEP以得到该频率下的正确时期：

In [41]: infl_q = infl.asfreq('Q-SEP', how='end')In [42]: infl_qOut[42]:1983Q1 0.0251984Q1 0.0451985Q1 0.0371986Q1 0.040Freq: Q-SEP

然后这个时间序列就可以被重索引了（使用前向填充以匹配gdp）：

In [43]: infl_q.reindex(gdp.index, method='ffill')Out[43]:1984Q2 0.0451984Q3 0.0451984Q4 0.0451985Q1 0.0371985Q2 0.0371985Q3 0.0371985Q4 0.037Freq: Q-SEP

时间和“最当前”数据选取

假设你有一个很长的盘中市场数据时间序列，现在希望抽取其中每天特定时间的价格数据。如果数据不规整（观测值没有精确地落在期望的时间点上），该怎么办？在实际工作当中，如果不够小心仔细的话，很容易导致错误的数据规整化。看看下面这个例子：

生成一个交易日内的日期范围和时间序列译注2In [44]: rng = pd.date_range('2012-06-01 09:30', '2012-06-01 15:59', freq='T')# 生成5天的时间点（9:30~15:59之间的值）In [45]: rng = rng.append([rng + pd.offsets.BDay(i) for i in range(1, 4)])In [46]: ts = Series(np.arange(len(rng), dtype=float), index=rng)In [47]: tsOut[47]:2012-06-01 09:30:00 02012-06-01 09:31:00 12012-06-01 09:32:00 22012-06-01 09:33:00 3...2012-06-06 15:56:00 15562012-06-06 15:57:00 15572012-06-06 15:58:00 15582012-06-06 15:59:00 1559Length: 1560

利用Python的datetime.time对象进行索引即可抽取出这些时间点上的值：

In [48]: from datetime import timeIn [49]: ts[time(10, 0)]Out[49]:2012-06-01 10:00:00 302012-06-04 10:00:00 4202012-06-05 10:00:00 8102012-06-06 10:00:00 1200

实际上，该操作用到了实例方法at_time（各时间序列以及类似的DataFrame对象都有）：

In [50]: ts.at_time(time(10, 0))Out[50]:2012-06-01 10:00:00 302012-06-04 10:00:00 4202012-06-05 10:00:00 8102012-06-06 10:00:00 1200

还有一个between_time方法，它用于选取两个Time对象之间的值：

In [51]: ts.between_time(time(10, 0), time(10, 1))Out[51]:2012-06-01 10:00:00 302012-06-01 10:01:00 312012-06-04 10:00:00 4202012-06-04 10:01:00 4212012-06-05 10:00:00 8102012-06-05 10:01:00 8112012-06-06 10:00:00 12002012-06-06 10:01:00 1201

正如之前提到的那样，可能刚好就没有任何数据落在某个具体的时间上（比如上午10点）。这时，你可能会希望得到上午10点之前最后出现的那个值：

将该时间序列的大部分内容随机设置为NAIn [53]: indexer = np.sort(np.random.permutation(len(ts))[700:])In [54]: irr_ts = ts.copy()In [55]: irr_ts[indexer] = np.nanIn [56]: irr_ts['2012-06-01 09:50':'2012-06-01 10:00']Out[56]:2012-06-01 09:50:00 NaN2012-06-01 09:51:00 NaN2012-06-01 09:52:00 222012-06-01 09:53:00 NaN2012-06-01 09:54:00 242012-06-01 09:55:00 NaN2012-06-01 09:56:00 262012-06-01 09:57:00 272012-06-01 09:58:00 282012-06-01 09:59:00 292012-06-01 10:00:00 NaN

如果将一组Timestamp传入asof方法，就能得到这些时间点处（或其之前最近）的有效值（非NA）。例如，我们构造一个日期范围（每天上午10点），然后将其传入asof：

In [57]: selection = pd.date_range('2012-06-01 10:00', periods=4, freq='B')In [58]: irr_ts.asof(selection)Out[58]:2012-06-01 10:00:00 292012-06-04 10:00:00 4192012-06-05 10:00:00 8102012-06-06 10:00:00 1198Freq: B

拼接多个数据源

在第7章中，我介绍了一些合并两个相关数据集的办法。在金融或经济领域中，还有另外几个经常出现的情况：

·在一个特定的时间点上，从一个数据源切换到另一个数据源。

·用另一个时间序列对当前时间序列中的缺失值“打补丁”。

·将数据中的符号（国家、资产代码等）替换为实际数据。

对于第一种情况，在特定时刻从一个时间序列切换到另一个，其实就是用pandas.concat将两个TimeSeries或DataFrame对象合并到一起：

In [59]: data1 = DataFrame(np.ones((6, 3), dtype=float), ...: columns=['a', 'b', 'c'], ...: index=pd.date_range('6/12/2012', periods=6))In [60]: data2 = DataFrame(np.ones((6, 3), dtype=float) * 2, ...: columns=['a', 'b', 'c'], ...: index=pd.date_range('6/13/2012', periods=6))In [61]: spliced = pd.concat([data1.ix[:'2012-06-14'], data2.ix['2012-06-15':]])In [62]: splicedOut[62]: a b c2012-06-12 1 1 12012-06-13 1 1 12012-06-14 1 1 12012-06-15 2 2 22012-06-16 2 2 22012-06-17 2 2 22012-06-18 2 2 2

再看另一个简单的例子，假设data1缺失了data2中存在的某个时间序列：

In [113]: data2 = DataFrame(np.ones((6, 4), dtype=float) * 2, ...: columns=['a', 'b', 'c', 'd'], ...: index=pd.date_range('6/13/2012', periods=6))In [64]: spliced = pd.concat([data1.ix[:'2012-06-14'], data2.ix['2012-06-15':]])In [65]: splicedOut[65]: a b c d2012-06-12 1 1 1 NaN2012-06-13 1 1 1 NaN2012-06-14 1 1 1 NaN2012-06-15 2 2 2 22012-06-16 2 2 2 22012-06-17 2 2 2 22012-06-18 2 2 2 2

combine_first可以引入合并点之前的数据，这样也就扩展了'd'项的历史：

In [66]: spliced_filled = spliced.combine_first(data2)In [67]: spliced_filledOut[67]: a b c d2012-06-12 1 1 1 NaN2012-06-13 1 1 1 22012-06-14 1 1 1 22012-06-15 2 2 2 22012-06-16 2 2 2 22012-06-17 2 2 2 22012-06-18 2 2 2 2

由于data2没有关于2012-06-12的数据，所以也就没有值被填充到那一天。

DataFrame也有一个类似的方法update，它可以实现就地更新。如果只想填充空洞，则必须传入overwrite=False才行：

In [68]: spliced.update(data2, overwrite=False)In [69]: splicedOut[69]: a b c d2012-06-12 1 1 1 NaN2012-06-13 1 1 1 22012-06-14 1 1 1 22012-06-15 2 2 2 22012-06-16 2 2 2 22012-06-17 2 2 2 22012-06-18 2 2 2 2

上面所讲的这些技术都可实现将数据中的符号替换为实际数据，但有时利用DataFrame的索引机制直接对列进行设置会更简单一些：

In [70]: cp_spliced = spliced.copy()In [71]: cp_spliced[['a', 'c']] = data1[['a', 'c']]In [72]: cp_splicedOut[72]: a b c d2012-06-12 1 1 1 NaN2012-06-13 1 1 1 22012-06-14 1 1 1 22012-06-15 1 2 1 22012-06-16 1 2 1 22012-06-17 1 2 1 22012-06-18 NaN 2 NaN 2

收益指数和累计收益

在金融领域中，收益（return）通常指的是某资产价格的百分比变化。我们来看看2011年到2012年间苹果公司的股票价格数据译注3：

In [73]: import pandas.io.data as webIn [74]: price = web.get_data_yahoo('AAPL', '2011-01-01')['Adj Close']In [75]: price[-5:]Out[75]:Date2012-07-23 603.832012-07-24 600.922012-07-25 574.972012-07-26 574.882012-07-27 585.16Name: Adj Close

对于苹果公司的股票（没有股息

译注4

），计算两个时间点之间的累计百分比回报只需计算价格的百分比变化即可：

In [76]: price['2011-10-03'] / price['2011-3-01'] - 1Out[76]: 0.072399874037388123

对于其他那些派发股息的股票，要计算你在某只股票上赚了多少钱就比较复杂了。不过，这里所使用的已调整收盘价已经对拆分和股息做出了调整。不管什么样的情况，通常都会先算出一个收益指数，它是一个表示单位投资（比如1美元）收益的时间序列。从收益指数中可以得出许多假设。例如，人们可以决定是否进行利润再投资。对于苹果公司的情况，我们可以利用cumprod计算出一个简单的收益指数：

In [77]: returns = price.pct_change()In [78]: ret_index = (1 + returns).cumprod()In [79]: ret_index[0] = 1 # 将第一个值设置为1In [80]: ret_indexOut[80]:Date2011-01-03 1.0000002011-01-04 1.0052192011-01-05 1.0134422011-01-06 1.012623...2012-07-24 1.8233462012-07-25 1.7446072012-07-26 1.7443342012-07-27 1.775526Length: 396

得到收益指数之后，计算指定时期内的累计收益就很简单了：

In [81]: m_returns = ret_index.resample('BM', how='last').pct_change()In [82]: m_returns['2012']Out[82]:Date2012-01-31 0.1271112012-02-29 0.1883112012-03-30 0.1052842012-04-30 -0.0259692012-05-31 -0.0107022012-06-29 0.0108532012-07-31 0.001986Freq: BM

当然了，就这个简单的例子而言（没有股息也没有其他需要考虑的调整），上面的结果也能通过重采样聚合（这里聚合为时期）从日百分比变化中计算得出：

In [83]: m_rets = (1 + returns).resample('M', how='prod', kind='period') - 1In [84]: m_rets['2012']Out[84]:Date2012-01 0.1271112012-02 0.1883112012-03 0.1052842012-04 -0.0259692012-05 -0.0107022012-06 0.0108532012-07 0.001986Freq: M

如果知道了股息的派发日和支付率，就可以将它们计入到每日总收益中，如下所示：

returns[dividend_dates] += dividend_pcts

译注1

：此处代码不完整，需要加载ch11的两个csv文件，然后稍作处理即可得到这里所需的素材。

译注2

：这里生成的只是索引，没有时间序列。

译注3：直接使用这段代码获取的数据会多很多，因没有截止日期，建议使用price=web.get_data_yahoo('AAPL','2011-01-01','2012-07-27')['Adj Close']。此外，由于这里获取的是Adj Close，所以数据本身也会有一些不同。

译注4

：现在已经派过股息了。

分组变换和分析

在第9章中，我们学习了组统计计算的基础知识，还学习了如何对数据集的分组应用自定义的变换函数。

下面以一组假想的股票投资组合为例。首先我随机生成1000个股票代码：

import random; random.seed(0)import stringN = 1000def rands(n): choices = string.ascii*uppercase return ''.join([random.choice(choices) for* in xrange(n)])tickers = np.array([rands(5) for _ in xrange(N)])

然后创建一个含有3列的DataFrame来承载这些假想数据，不过只选择部分股票组成该投资组合：

M = 500df = DataFrame({'Momentum' : np.random.randn(M) / 200 + 0.03, 'Value' : np.random.randn(M) / 200 + 0.08, 'ShortInterest' : np.random.randn(M) / 200 - 0.02}, index=tickers[:M])

接下来，我们为这些股票随机创建一个行业分类。为了简单起见，我只选用了两个行业，并将映射关系保存在Series中：

ind_names = np.array(['FINANCIAL', 'TECH'])sampler = np.random.randint(0, len(ind_names), N)industries = Series(ind_names[sampler], index=tickers, name='industry')

现在，我们就可以根据行业分类进行分组并执行分组聚合和变换了：

In [90]: by_industry = df.groupby(industries)In [91]: by_industry.mean()Out[91]: Momentum ShortInterest ValueindustryFINANCIAL 0.029485 -0.020739 0.079929TECH 0.030407 -0.019609 0.080113In [92]: by_industry.describe()Out[92]: Momentum ShortInterest ValueindustryFINANCIAL count 246.000000246.000000 246.000000 mean 0.029485 -0.020739 0.079929 std 0.004802 0.004986 0.004548 min 0.017210 -0.036997 0.067025 25% 0.026263 -0.024138 0.076638 50% 0.029261 -0.020833 0.079804 75% 0.032806 -0.017345 0.082718 max 0.045884 -0.006322 0.093334TECH count 254.000000 254.000000 254.000000 mean 0.030407 -0.019609 0.080113 std 0.005303 0.005074 0.004886 min 0.016778 -0.032682 0.065253 25% 0.026456 -0.022779 0.076737 50% 0.030650 -0.019829 0.080296 75% 0.033602 -0.016923 0.083353 max 0.049638 -0.003698 0.093081

要对这些按行业分组的投资组合进行各种变换，我们可以编写自定义的变换函数。例如行业内标准化处理，它广泛用于股票资产投资组合的构建过程：

行业内标准化处理def zscore(group): return (group - group.mean()) / group.std()df_stand = by_industry.apply(zscore)

这样处理之后，各行业的平均值为0，标准差为1：

In [94]: df_stand.groupby(industries).agg(['mean', 'std'])Out[94]: Momentum ShortInterest Value mean std mean std mean stdindustryFINANCIAL 0 1 0 1 0 1TECH -0 1 -0 1 -0 1

内置变换函数（如rank）的用法会更简洁一些：

行业内降序排名In [95]: ind_rank = by_industry.rank(ascending=False)In [96]: ind_rank.groupby(industries).agg(['min', 'max'])Out[96]: Momentum ShortInterest Value min max min max min maxindustryFINANCIAL 1 246 1 246 1 246TECH 1 254 1 254 1 254

在股票投资组合的定量分析中，“排名和标准化”是一种很常见的变换运算组合。通过将rank和zscore链接在一起即可完成整个变换过程，就像下面这样：

行业内排名和标准化In [97]: by_industry.apply(lambda x: zscore(x.rank()))Out[97]:Index: 500 entries, VTKGN to PTDQEData columns:Momentum 500 non-null

valuesShortInterest 500 non-null valuesValue 500 non-null valuesdtypes: float64(3)

分组因子暴露

因子分析（factor analysis）是投资组合定量管理中的一种技术。投资组合的持有量和性能（收益与损失）可以被分解为一个或多个表示投资组合权重的因子（风险因子就是其中之一）。例如，某只股票的价格与某个基准（比如标准普尔500指数）的协动性被称作其贝塔风险系数（beta，一种常见的风险因子）。下面以一个人为构成的投资组合为例进行讲解，它由三个随机生成的因子（通常称为因子载荷）和一些权重构成：

from numpy.random import randfac1, fac2, fac3 = np.random.rand(3, 1000)ticker_subset = tickers.take(np.random.permutation(N)[:1000])# 因子加权和以及噪声port = Series(0.7 *fac1 - 1.2* fac2 + 0.3 * fac3 + rand(1000), index=ticker_subset)factors = DataFrame({'f1': fac1, 'f2': fac2, 'f3': fac3}, index=ticker_subset)

各因子与投资组合之间的矢量相关性可能说明不了什么问题：

In [99]: factors.corrwith(port)Out[99]:f1 0.402377f2 -0.680980f3 0.168083

计算因子暴露的标准方式是最小二乘回归。使用pandas.ols（将factors作为解释变量）即可计算出整个投资组合的暴露：

In [100]: pd.ols(y=port, x=factors).betaOut[100]:f1 0.761789f2 -1.208760f3 0.289865intercept 0.484477

不难看出，由于没有给投资组合添加过多的随机噪声，所以原始的因子权重基本上可算是恢复出来了。还可以通过groupby计算各行业的暴露量。为了达到这个目的，我先编写了一个函数，如下所示：

def beta_exposure(chunk, factors=None): return pd.ols(y=chunk, x=factors).beta

然后根据行业进行分组，并应用该函数，传入因子载荷的DataFrame：

In [102]: by_ind = port.groupby(industries)In [103]: exposures = by_ind.apply(beta_exposure, factors=factors)In [104]: exposures.unstack()Out[104]: f1 f2 f3 interceptindustryFINANCIAL 0.790329 -1.182970 0.275624 0.455569TECH 0.740857 -1.232882 0.303811 0.508188

十分位和四分位分析

基于样本分位数的分析是金融分析师们的另一个重要工具。例如，股票投资组合的性能可以根据各股的市盈率被划分入四分位（四个大小相等的块）。通过pandas.qcut和groupby可以非常轻松地实现分位数分析。

在下面这个例子中，我们利用跟随策略或动量交易策略通过SPY交易所交易基金买卖标准普尔500指数。你可以从Yahoo!Finance下载价格历史：

In [105]: import pandas.io.data as webIn [106]: data = web.get_data_yahoo('SPY', '2006-01-01')译注5In [107]: dataOut[107]:DatetimeIndex: 1655 entries, 2006-01-03 00:00:00 to 2012-07-27 00:00:00Data columns:Open 1655 non-null valuesHigh 1655 non-null valuesLow 1655 non-null valuesClose 1655 non-null valuesVolume 1655 non-null valuesAdj Close 1655 non-null valuesdtypes: float64(5), int64(1)

接下来计算日收益率，并编写一个用于将收益率变换为趋势信号（通过滞后移动形成）的函数：

px = data['Adj Close']returns = px.pct_change()def to_index(rets): index = (1 + rets).cumprod() first_loc = max(index.notnull().argmax() - 1, 0) index.values[first_loc] = 1 return indexdef trend_signal(rets, lookback, lag): signal = pd.rolling_sum(rets, lookback, min_periods=lookback - 5) return signal.shift(lag)

通过该函数，我们可以（单纯地）创建和测试一种根据每周五动量信号进行交易的交易策略。

In [109]: signal = trend_signal(returns, 100, 3)In [110]: trade_friday = signal.resample('W-FRI').resample('B', fill_method='ffill')In [111]: trade_rets = trade_friday.shift(1) * returns

然后将该策略的收益率转换为一个收益指数，并绘制一张图表（如图11-1所示）：

In [112]: to_index(trade_rets).plot()

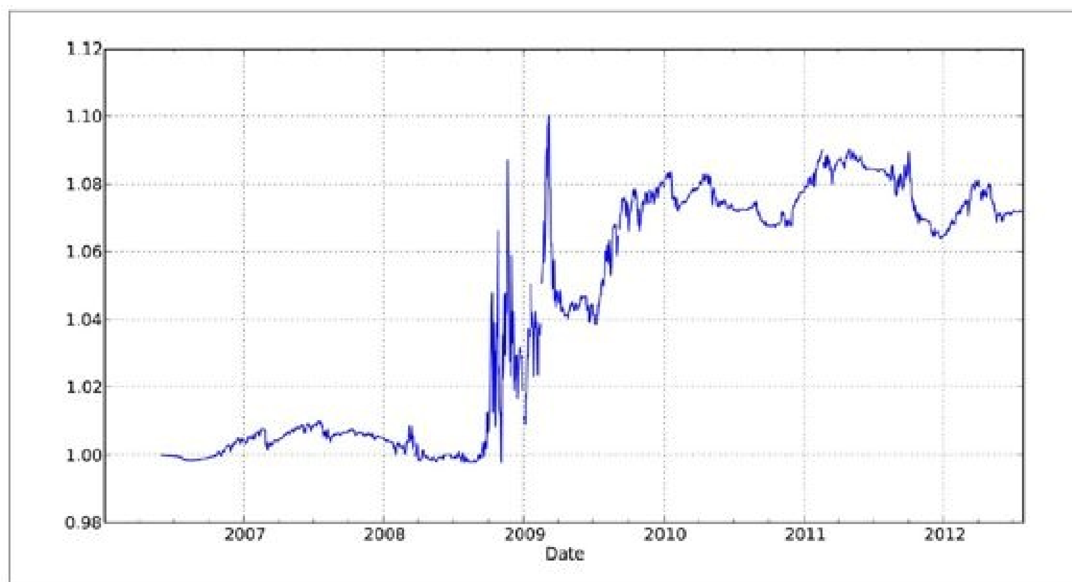


图11-1：SPY动量策略收益指数

假如你希望将该策略的性能按不同大小的交易期波幅进行划分。年度标准差是计算波幅的一种简单办法，我们可以通过计算夏普比率来观察不同波动机制下的风险收益率：

vol = pd.rolling_std(returns, 250, min_periods=200) *np.sqrt(250)def sharpe(rets, ann=250): return rets.mean() / rets.std()* np.sqrt(ann)

现在，利用qcut将vol划分为四等份，并用sharpe进行聚合：

In [114]: trade_rets.groupby(pd.qcut(vol, 4)).agg(sharpe)Out[114]:[0.0955, 0.16] 0.490051(0.16, 0.188] 0.482788(0.188, 0.231] -0.731199(0.231, 0.457] 0.570500

这个结果说明，该策略在波幅最高时性能最好。

译注5

：跟前面说的一样，这里最好还是加上截止日期，否则数据会比书上介绍的多。

更多示例应用

本节介绍一些其他的例子。

信号前沿分析

在本小节中，将介绍一种简化的截面动量投资组合，并告诉你如何得到模型参数化网格。首先，我将金融和技术领域中的几只股票做成一个投资组合，并加载它们的历史价格数据：

names = ['AAPL', 'GOOG', 'MSFT', 'DELL', 'GS', 'MS', 'BAC', 'C']def get_px(stock, start, end): return web.get_data_yahoo(stock, start, end)['Adj Close']px = DataFrame({n: get_px(n, '1/1/2009', '6/1/2012') for n in names})

我们可以轻松绘制每只股票的累计收益（如图11-2所示）：

In [117]: px = px.asfreq('B').fillna(method='pad')In [118]: rets = px.pct_change()In [119]: ((1 + rets).cumprod() - 1).plot()

对于投资组合的构建，我们要计算特定回顾期的动量，然后按降序排列并标准化：

def calc_mom(price, lookback, lag): mom_ret = price.shift(lag).pct_change(lookback) ranks = mom_ret.rank(axis=1, ascending=False) demeaned = ranks - ranks.mean(axis=1) return demeaned / demeaned.std(axis=1)

利用这个变换函数，我们再编写一个对策略进行事后检验的函数：通过指定回顾期和持有期（买卖之间的日数）计算投资组合整体的夏普比率。

compound = lambda x : (1 + x).prod() - 1daily_sr = lambda x: x.mean() / x.std()def strat_sr(prices, lb, hold): # 计算投资组合权重 freq = '%dB' % hold port = calc_mom(prices, lb, lag=1) daily_rets = prices.pct_change() # 计算投资组合收益 port =

port.shift(1).resample(freq, how='first') returns = daily_rets.resample(freq, how=compound) port_rets = (port *returns).sum(axis=1) return daily_sr(port_rets)* np.sqrt(252 / hold)

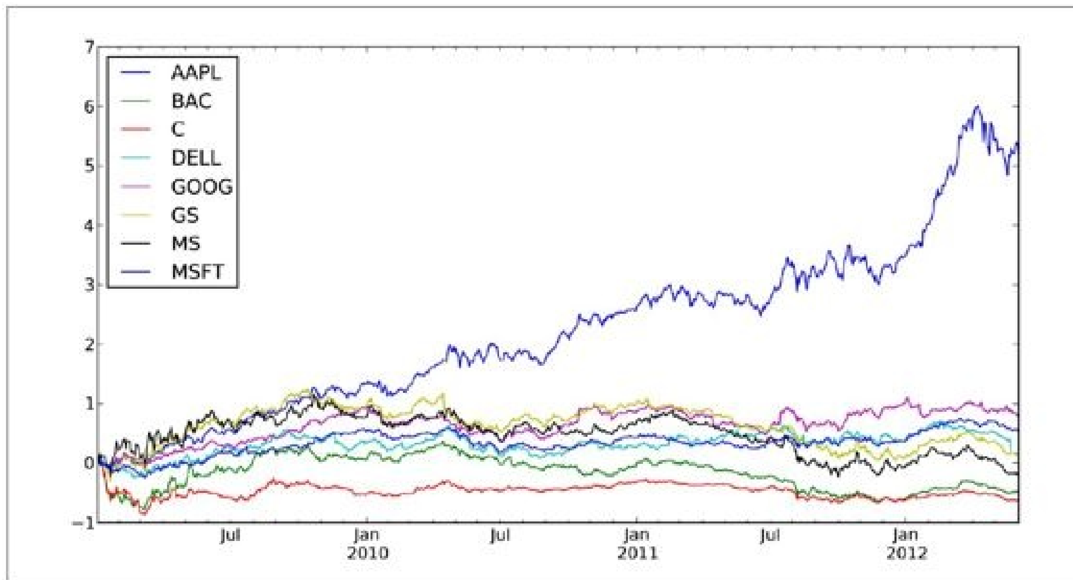


图11-2：每只股票的累计收益

通过价格数据以及一对参数组合调用该函数将会得到一个标量值：

In [122]: strat_sr(px, 70, 30)Out[122]: 0.27421582756800583

然后对参数网格（即多对参数组合）应用strat_sr函数，并将结果保存在一个defaultdict中，最后再将全部结果放进一个DataFrame中：

from collections import defaultdictlookbacks = range(20, 90, 5)holdings = range(20, 90, 5)dd = defaultdict(dict)for lb in lookbacks: for hold in holdings: dd[lb][hold] = strat_sr(px, lb, hold)ddf = DataFrame(dd)ddf.index.name = 'Holding Period'ddf.columns.name = 'Lookback Period'

为了便于观察，我们可以将该结果图形化。下面这个函数会利用matplotlib生成一张带有装饰物

译注6

的热图（heatmap）：

import matplotlib.pyplot as pltdef heatmap(df, cmap=plt.cm.gray_r): fig = plt.figure() ax = fig.add_subplot(111) axim = ax.imshow(df.values, cmap=cmap, interpolation='nearest') ax.set_xlabel(df.columns.name) ax.set_xticks(np.arange(len(df.columns))) ax.set_xticklabels(list(df.columns)) ax.set_ylabel(df.index.name) ax.set_yticks(np.arange(len(df.index))) ax.set_yticklabels(list(df.index)) plt.colorbar(axim)

对事后检验结果调用该函数，就会得到图11-3：

In [125]: heatmap(ddf)

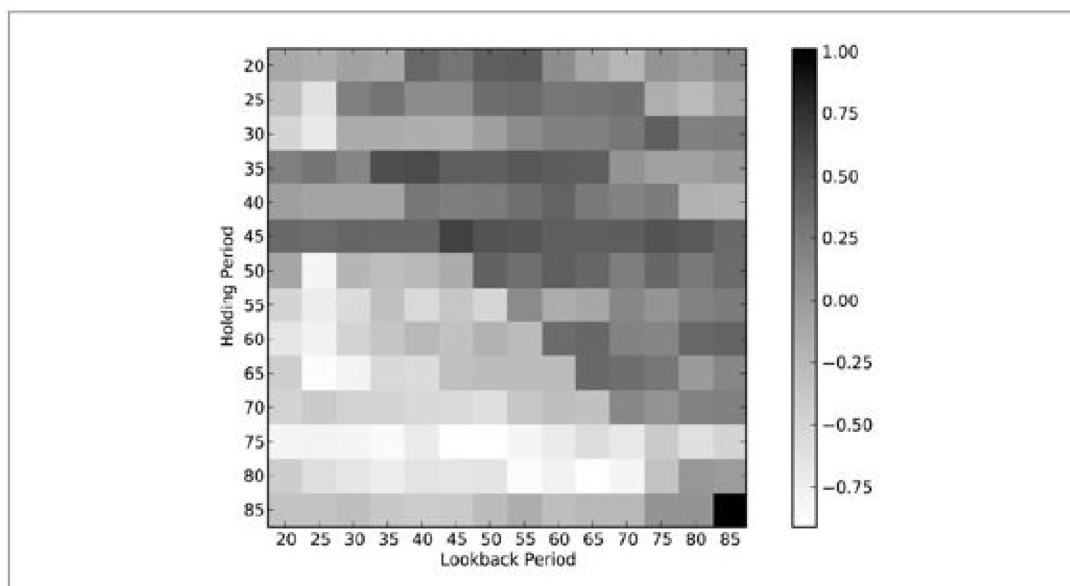


图11-3：动量策略各种回顾期和持有期的夏普比率热图（越高越好）

期货合约转仓

期货是一种无所不在的衍生品合约。它是一种在指定日期交收指定资产（比如石油、黄金或FTSE100指数的股份）的约定。在实践中，由于期货合约具有限时性，对（股票、货币、商品、债券以及其他资产类）期货合约的建模和交易是很复杂的。例如，对于某种期货（比如银或铜期货），在给定时间点，可能有多个到期时间不同的合约被交易。一般来说，下一个期满的期货合约（即近期合约）将是最具流动性的（成交量最高和买卖差价最低）。

通过一个表示盈亏（始终持有近期合约）的连续的收益指数即可轻松实现建模和预测。从一份到期合约过渡到下一期（或更远的）合约称为转仓。通过单个期货合约数据构建连续序列并不简单，而且一般都需要深入了解市场以及交易方面的知识才行。例如，你该何时以及如何快速卖出到期合约并买入下期合约？本节我所描述的就是这样的一个过程。

首先，我用SPY交易所交易基金的部分价格作为标准普尔500指数的代理：

In [127]: import pandas.io.data as web# 标准普尔500指数的近似价格In [128]: px = web.get_data_yahoo('SPY')['Adj Close'] * 10In [129]: pxOut[129]:Date2011-08-01 1261.02011-08-02 1228.82011-08-03 1235.5...2012-07-25 1339.62012-07-26 1361.72012-07-27 1386.8Name: Adj Close, Length: 251

现在，稍微做一些设置。我在一个Series中放了两份标准普尔500指数期货合约及其到期日期：

from datetime import datetimeexpiry = {'ESU2': datetime(2012, 9, 21), 'ESZ2': datetime(2012, 12, 21)}expiry = Series(expiry).order()

expiry现在应该是这个样子的：

In [131]: expiryOut[131]:ESU2 2012-09-21 00:00:00ESZ2 2012-12-21 00:00:00

然后，我用Yahoo!Finance的价格以及一个随机漫步和一些噪声来模拟这两份合约未来的走势：

np.random.seed(12347)N = 200walk = (np.random.randint(0, 200, size=N) - 100) *0.25perturb = (np.random.randint(0, 20, size=N) - 10)* 0.25walk = walk.cumsum()rng = pd.date_range(px.index[0], periods=len(px) + N, freq='B')near = np.concatenate([px.values, px.values[-1] + walk])far = np.concatenate([px.values, px.values[-1] + walk + perturb])prices = DataFrame({'ESU2': near, 'ESZ2': far}, index=rng)

这样，prices就有了关于这两个合约的时间序列：

In [133]: prices.tail()Out[133]: ESU2 ESZ22013-04-16 1416.05 1417.802013-04-17 1402.30 1404.552013-04-18 1410.30 1412.052013-04-19 1426.80 1426.052013-04-22 1406.80 1404.55

将多个时间序列合并为单个连续序列的一个办法是构造一个加权矩阵。活动合约的权重应该设为1，直到期满为止。在那个时候，你必须决定一个转仓约定。下面这个函数可以计算一个加权矩阵（权重根据到期前的期数减少而线性衰减）：

def get_roll_weights(start, expiry, items, roll_periods=5): # start : 用于计算加权矩阵的第一天 # expiry : 由“合约代码 -> 到期日期”组成的序列 # items : 一组合约名称 dates = pd.date_range(start, expiry[-1], freq='B') weights = DataFrame(np.zeros((len(dates), len(items))), index=dates, columns=items) prev_date = weights.index[0] for i, (item, ex_date) in enumerate(expiry.iteritems()): if i < len(expiry) - 1: weights.ix[prev_date:ex_date - pd.offsets.BDay(), item] = 1 roll_rng = pd.date_range(end=ex_date - pd.offsets.BDay(), periods=roll_periods + 1, freq='B') decay_weights = np.linspace(0, 1, roll_periods + 1) weights.ix[roll_rng, item] = 1 - decay_weights weights.ix[roll_rng, expiry.index[i + 1]] = decay_weights else: weights.ix[prev_date:, item] = 1 prev_date = ex_date return weights

快到ESU2到期日的那几天的权重如下所示：

In [135]: weights = get_roll_weights('6/1/2012', expiry, prices.columns)In [136]: weights.ix['2012-09-12':'2012-09-21']Out[136]: ESU2 ESZ22012-09-12 1.0 0.02012-09-13 1.0 0.02012-09-14 0.8 0.22012-09-17 0.6 0.42012-09-18 0.4 0.62012-09-19 0.2 0.82012-09-20 0.0 1.02012-09-21 0.0 1.0

最后，转仓期货收益就是合约收益的加权和：

In [137]: rolled_returns = (prices.pct_change() * weights).sum(1)

移动相关系数与线性回归

动态模型在金融建模工作中扮演着重要的角色，因为它们可用于模拟历史时期中的交易决策。移动窗口和指数加权时间序列函数就是用于处理动态模型的工具。

相关系数是观察两个资产时间序列的变化的协动性的一种手段。pandas的rolling_corr函数可以根据两个收益序列计算出移动窗口相关系数。首先，我从Yahoo!Finance加载一些价格序列，并计算每日收益率：

aapl = web.get_data_yahoo('AAPL', '2000-01-01')['Adj Close']msft = web.get_data_yahoo('MSFT', '2000-01-01')['Adj Close']aapl_rets = aapl.pct_change()msft_rets = msft.pct_change()

然后，我计算一年期移动相关系数并绘制图表（如图11-4所示）：

In [140]: pd.rolling_corr(aapl_rets, msft_rets, 250).plot()

两个资产之间的相关系数存在一个问题，即它不能捕获波动性差异。最小二乘回归提供了另一种对一个变量与一个或多个其他预测变量之间动态关系的建模办法。

In [142]: model = pd.ols(y=aapl_rets, x={'MSFT': msft_rets}, window=250)In [143]: model.betaOut[143]:DatetimeIndex: 2913 entries, 2000-12-28 00:00:00 to 2012-07-27 00:00:00Data columns:MSFT 2913 non-null valuesintercept 2913 non-null valuesdtypes: float64(2)In [144]: model.beta['MSFT'].plot()

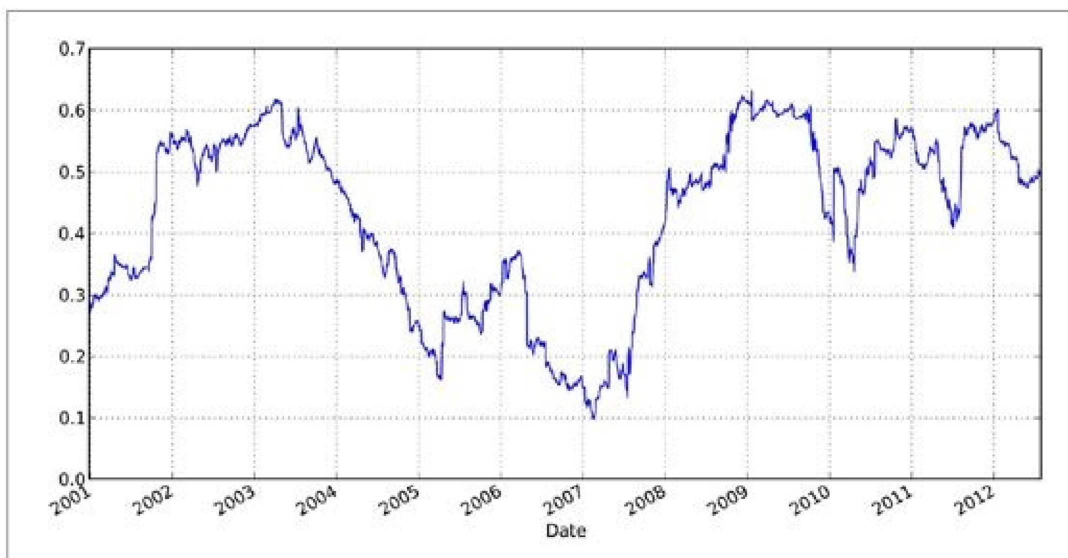


图11-4：苹果与微软的一年期相关系数

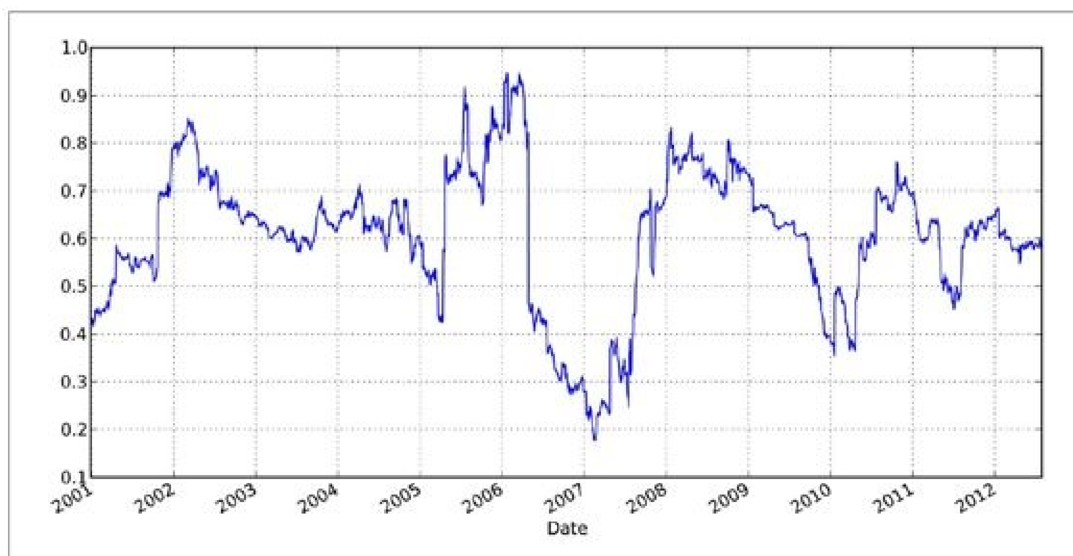


图11-5：苹果对微软一年期beta（OLS回归系数）

pandas的ols函数实现了静态和动态（扩展或移动窗口）的最小二乘回归。有关统计学和计量经济学的复杂模型的更多信息，请参考statsmodels项目（http://statsmodels.sourceforge.net）。

译注6

："装饰物"就是图例、标题之类的"配角"元素。

第12章　NumPy高级应用

ndarray对象的内部机理

NumPy的ndarray提供了一种将同质数据块（可以是连续或跨越译注1的，稍后将详细讲解）解释为多数组对象的方式。正如你之前所看到的那样，数据类型（dtype）决定了数据的解释方式，比如浮点数、整数、布尔值等。

ndarray如此强大的部分原因是所有数组对象都是数据块的一个跨度视图（strided view）。你可能想知道数组视图arr[::2,::-1]不复制任何数据的原因是什么。简单地说，ndarray不只是一块内存和一个dtype，它还有跨度信息，这使得数组能以各种步幅（step size）在内存中移动

译注2

。更准确地讲，ndarray内部由以下内容组成：

·一个指向数组（一个系统内存块）的指针。

·数据类型或dtype。

·一个表示数组形状（shape）的元组，例如，一个10×5的数组，其形状为(10,5)。

```
In [8]: np.ones((10, 5)).shape       Out[8]: (10, 5)
```

·一个跨度元组（stride），其中的整数指的是为了前进到当前维度下一个元素需要“跨过”的字节数，例如，一个典型的（C顺序，稍后将详细讲解）3×4×5的float64（8个字节）数组，其跨度为(160,40,8)。

```
In [9]: np.ones((3, 4, 5), dtype=np.float64).strides      Out[9]: (160, 40, 8)
```

虽然NumPy用户很少会对数组的跨度信息感兴趣，但它们却是构建非复制式数组视图的重要因素。跨度甚至可以是负数，这样会使数组在内存中后向移动，比如在切片obj[::-1]或obj[:,::-1]中就是这样的。

图12-1简单地说明了ndarray的内部结构。

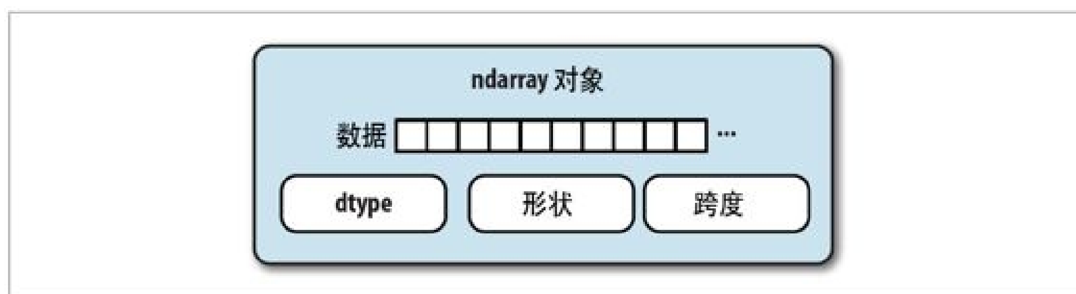


图12-1：NumPy的ndarray对象

NumPy数据类型体系

你可能偶尔需要检查数组中所包含的是否是整数、浮点数、字符串或Python对象。因为浮点数的种类很多，判断dtype是否属于某个大类的工作非常繁琐。幸运的是，dtype都有一个超类（比如np.integer和np.floating），它们可以跟np.issubdtype函数结合使用：

In [10]: ints = np.ones(10, dtype=np.uint16)In [11]: floats = np.ones(10, dtype=np.float32)In [12]: np.issubdtype(ints.dtype, np.integer)Out[12]: TrueIn [13]: np.issubdtype(floats.dtype, np.floating)Out[13]: True

调用dtype的mro方法即可查看其所有的父类：

In [14]: np.float64.mro()Out[14]:[numpy.float64, numpy.floating, numpy.inexact, numpy.number, numpy.generic, float, object]

大部分NumPy用户完全不需要了解这些知识，但是这些知识偶尔还是能派上用场的。图12-2说明了dtype体系以及父子类关系注1。

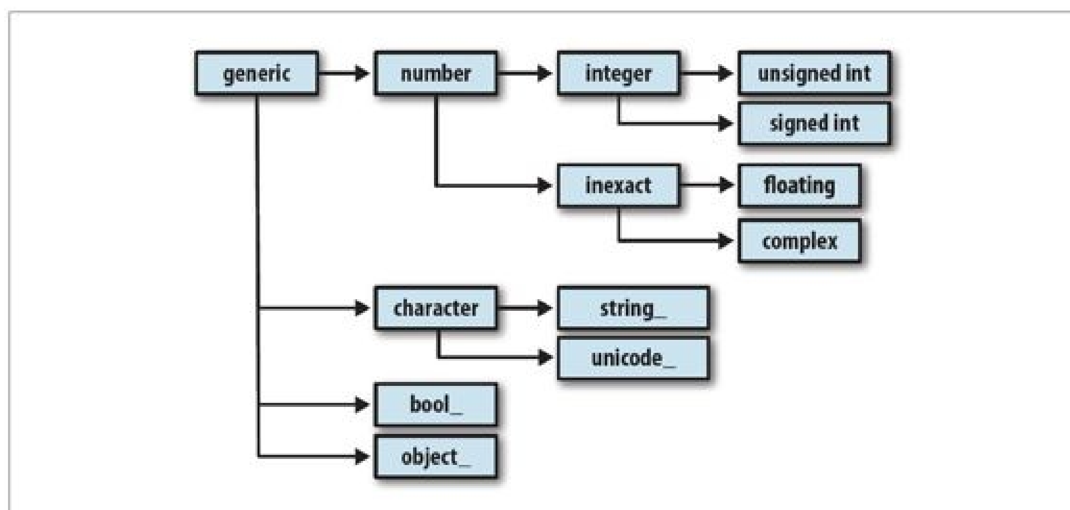


图12-2：NumPy的dtype体系

译注1

：也就是非连续存储。

译注2：数组本身不能移动，这里实际上说的是指针。

注1

：有些dtype的名称后面带有下划线，这是了避免NumPy类型和Python类型之间的变量名冲突。

高级数组操作

除花式索引、切片、布尔条件取子集等操作之外，数组的操作方式还有很多。虽然pandas中的高级函数可以处理数据分析工作中的许多重型任务，但有时你还是需要编写一些在现有库找不到的数据算法。

数组重塑

鉴于我们已经学过的有关NumPy数组的知识，当你知道“无需复制任何数据，数组就能从一个形状转换为另一个形状”时应该会感到有一点吃惊。只需向数组的实例方法reshape传入一个表示新形状的元组即可实现该目的。假设有一个一维数组，我们希望将其重新排列为一个矩阵：

In [15]: arr = np.arange(8)In [16]: arrOut[16]: array([0, 1, 2, 3, 4, 5, 6, 7])In [17]: arr.reshape((4, 2))Out[17]:array([[0, 1], [2, 3], [4, 5], [6, 7]])

多维数组也能被重塑：

In [18]: arr.reshape((4, 2)).reshape((2, 4))Out[18]:array([[0, 1, 2, 3], [4, 5, 6, 7]])

作为参数的形状的其中一维可以是－1，它表示该维度的大小由数据本身推断而来：

In [19]: arr = np.arange(15) In [20]: arr.reshape((5, -1)) Out[20]: array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11], [12, 13, 14]])

由于数组的shape属性是一个元组，因此它也可以被传入reshape：

In [21]: other_arr = np.ones((3, 5))In [22]: other_arr.shapeOut[22]: (3, 5)In [23]: arr.reshape(other_arr.shape)Out[23]:array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14]])

与reshape将一维数组转换为多维数组的运算过程相反的运算通常称为扁平化（flattening）或散开（raveling）：

In [24]: arr = np.arange(15).reshape((5, 3)) In [25]: arr Out[25]: array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11], [12, 13, 14]])In [26]: arr.ravel()Out[26]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])

如果没有必要，ravel不会产生源数据的副本（下面将详细介绍）。flatten方法的行为类似于ravel，只不过它总是返回数据的副本：

In [27]: arr.flatten()Out[27]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])

数组可以被重塑或散开为别的顺序。这对NumPy新手来说是一个比较微妙的问题，所以在下一小节中我们将专门讲解这个问题。

C和Fortran顺序

与其他科学计算环境相反（如R和MATLAB），NumPy允许你更为灵活地控制数据在内存中的布局。默认情况下，NumPy数组是按行优先顺序创建的。在空间方面，这就意味着，对于一个二维数组，每行中的数据项是被存放在相邻内存位置上的。另一种顺序是列优先顺序，它意味着（猜到了吧）每列中的数据项是被存放在相邻内存位置上的。

由于一些历史原因，行和列优先顺序又分别称为C和Fortran顺序。在FORTRAN 77中（前辈们的语言），矩阵全都是列优先的。

像reshape和reval这样的函数，都可以接受一个表示数组数据存放顺序的order参数。一般可以是'C'或'F'（还有'A'和'K'等不常用的选项，具体请参考NumPy的文档）。图12-3对此进行了说明。

In [28]: arr = np.arange(12).reshape((3, 4))In [29]: arrOut[29]:array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]])In [30]: arr.ravel()Out[30]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])In [31]: arr.ravel('F')Out[31]: array([0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11])

二维或更高维数组的重塑过程比较令人费解。C和Fortran顺序的关键区别就是维度的行进顺序：

·C/行优先顺序：先经过更高的维度（例如，轴1会先于轴0被处理）。

·Fortran/列优先顺序：后经过更高的维度（例如，轴0会先于轴1被处理）。

数组的合并和拆分

numpy.concatenate可以按指定轴将一个由数组组成的序列（如元组、列表等）连接到一起。

In [32]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])In [33]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])In [34]: np.concatenate([arr1, arr2], axis=0)Out[34]:array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])In [35]: np.concatenate([arr1, arr2], axis=1)Out[35]:array([[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11, 12]])

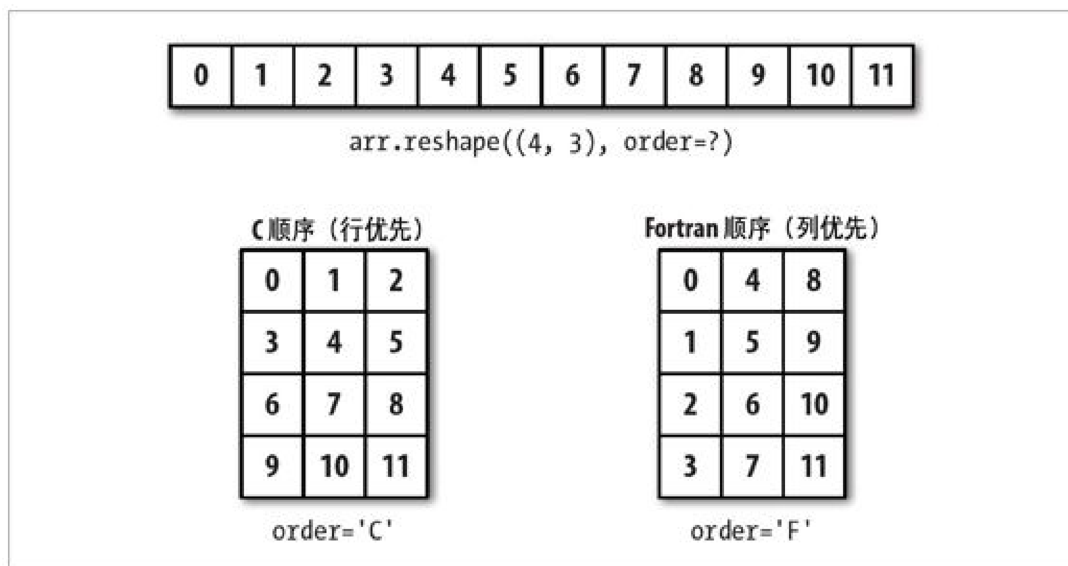


图12-3：按C（行优先）或Fortran（列优先）顺序进行重塑

对于常见的连接操作，NumPy提供了一些比较方便的方法（如vstack和hstack）。因此，上面的运算还可以表达为：

In [36]: np.vstack((arr1, arr2)) In [37]: np.hstack((arr1, arr2))Out[36]: Out[37]:array([[1, 2, 3], array([[1, 2, 3, 7, 8, 9], [4, 5, 6], [4, 5, 6, 10, 11, 12]]) [7, 8, 9], [10, 11, 12]])

与此相反，split用于将一个数组沿指定轴拆分为多个数组：

In [38]: from numpy.random import randnIn [39]: arr = randn(5, 2) In [40]: arr Out[40]: array([[0.1689, 0.3287], [0.4703, 0.8989], [0.1535, 0.0243], [-0.2832, 1.1536], [0.2707, 0.8075]])In [41]: first, second, third = np.split(arr, [1, 3])In [42]: firstOut[42]: array([[0.1689, 0.3287]])In [43]: second In [44]: thirdOut[43]: Out[44]:array([[0.4703, 0.8989], array([[-0.2832, 1.1536], [0.1535, 0.0243]]) [0.2707, 0.8075]])

表12-1中列出了所有关于数组连接和拆分的函数，其中有些是专门为了方便常见的连接运算而提供的。

表12-1：数组连接函数[译注3]

函数	说明
concatenate	最一般化的连接，沿一条轴连接一组数组
vstack、row_stack	以面向行的方式对数组进行堆叠（沿轴0）
hstack	以面向列的方式对数组进行堆叠（沿轴1）
column_stack	类似于hstack，但是会先将一维数组转换为二维列向量
dstack	以面向“深度”的方式对数组进行堆叠（沿轴2）
split	沿指定轴在指定的位置拆分数组
hsplit、vsplit、dsplit	split的便捷化函数，分别沿轴0、轴1、轴2进行拆分

译注3：这里面还有拆分函数。

堆叠辅助类：r和c

NumPy命名空间中有两个特殊的对象——r和c，它们可以使数组的堆叠操作更为简洁：

In [45]: arr = np.arange(6)In [46]: arr1 = arr.reshape((3, 2))In [47]: arr2 = randn(3, 2)In [48]: np.r[arr1, arr2] In [49]: np.c[np.r_[arr1, arr2], arr]Out[48]: Out[49]:array([[0. , 1.], array([[0. , 1. , 0.], [2. , 3.], [2. , 3. , 1.], [4. , 5.], [4. , 5. , 2.], [0.7258, -1.5325], [0.7258, -1.5325, 3.], [-0.4696, -0.2127], [-0.4696, -0.2127, 4.], [-0.1072, 1.2871]]) [-0.1072, 1.2871, 5.]])

此外，它还可以将切片翻译为数组：

In [50]: np.c_[1:6, -10:-5]Out[50]:array([[1, -10], [2, -9], [3, -8], [4, -7], [5, -6]])

r和c的具体功能请参考其文档。

元素的重复操作：tile和repeat

注意： 跟其他流行的数组编程语言（如MATLAB）不同，NumPy中很少需要对数组进行重复（replicate）

译注4

。这主要是因为广播（broadcasting，我们将在下一节中讲解该技术）能更好地满足该需求。

对数组进行重复以产生更大数组的工具主要是repeat和tile这两个函数。repeat会将数组中的各元素重复一定次数，从而产生一个更大的数组：

In [51]: arr = np.arange(3)In [52]: arr.repeat(3)Out[52]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])

默认情况下，如果传入的是一个整数，则各元素就都会重复那么多次。如果传入的是一组整数，则各元素就可以重复不同的次数：

In [53]: arr.repeat([2, 3, 4])Out[53]: array([0, 0, 1, 1, 1, 2, 2, 2, 2])

对于多维数组，还可以让它们的元素沿指定轴重复。

In [54]: arr = randn(2, 2)In [55]: arrOut[55]:array([[0.7157, -0.6387], [0.3626, 0.849]])In [56]: arr.repeat(2, axis=0)Out[56]:array([[0.7157, -0.6387], [0.7157, -0.6387], [0.3626, 0.849], [0.3626, 0.849]])

注意，如果没有设置轴向，则数组会被扁平化，这可能不会是你想要的结果。同样，在对多维进行重复时，也可以传入一组整数，这样就会使各切片重复不同的次数：

In [57]: arr.repeat([2, 3], axis=0)Out[57]:array([[0.7157, -0.6387], [0.7157, -0.6387], [0.3626, 0.849], [0.3626, 0.849], [0.3626, 0.849]])In [58]: arr.repeat([2, 3], axis=1)Out[58]:array([[0.7157, 0.7157, -0.6387, -0.6387, -0.6387], [0.3626, 0.3626, 0.849 , 0.849 , 0.849]])

tile的功能是沿指定轴向堆叠数组的副本。你可以形象地将其想象成“铺瓷砖”：

In [59]: arrOut[59]:array([[0.7157, -0.6387], [0.3626, 0.849]])In [60]: np.tile(arr, 2)Out[60]:array([[0.7157, -0.6387, 0.7157, -0.6387], [0.3626, 0.849 , 0.3626, 0.849]])

第二个参数是瓷砖的数量。对于标量，瓷砖是水平铺设的，而不是垂直铺设。它可以是一个表示“铺设”布局的元组：

In [61]: arrOut[61]:array([[0.7157, -0.6387], [0.3626, 0.849]])In [62]: np.tile(arr, (2, 1)) In [63]: np.tile(arr, (3, 2))Out[62]: Out[63]:array([[0.7157, -0.6387], array([[0.7157, -0.6387, 0.7157, -0.6387], [0.3626, 0.849], [0.3626, 0.849 , 0.3626, 0.849], [0.7157, -0.6387], [0.7157, -0.6387, 0.7157, -0.6387], [0.3626, 0.849]]) [0.3626, 0.849 , 0.3626, 0.849], [0.7157, -0.6387, 0.7157, -0.6387], [0.3626, 0.849 , 0.3626, 0.849]])

花式索引的等价函数：take和put

在第4章中我们讲过，获取和设置数组子集的一个办法是通过整数数组使用花式索引：

In [64]: arr = np.arange(10) * 100In [65]: inds = [7, 1, 2, 6] In [66]: arr[inds] Out[66]: array([700, 100, 200, 600])

ndarray有两个方法专门用于获取和设置单个轴向上的选区：

In [67]: arr.take(inds)Out[67]: array([700, 100, 200, 600])In [68]: arr.put(inds, 42)In [69]: arrOut[69]: array([0, 42, 42, 300, 400, 500, 42, 42, 800, 900])In [70]: arr.put(inds, [40, 41, 42, 43])In [71]: arrOut[71]: array([0, 41, 42, 300, 400, 500, 43, 40, 800, 900])

要在其他轴上使用take，只需传入axis关键字即可：

In [72]: inds = [2, 0, 2, 1]In [73]: arr = randn(2, 4)In [74]: arrOut[74]:array([[-0.8237, 2.6047, -0.4578, -1.], [2.3198, -1.0792, 0.518 , 0.2527]])In [75]: arr.take(inds, axis=1)Out[75]:array([[-0.4578, -0.8237, -0.4578, 2.6047], [0.518 , 2.3198, 0.518 , -1.0792]])

put不接受axis参数，它只会在数组的扁平化版本（一维，C顺序）上进行索引（这一点今后应该是会有所改善的）。因此，在需要用其他轴向的索引设置元素时，最好还是使用花式索引。

注意：直到编写本书时为止，take和put函数的性能通常要比花式索引好得多。我认为这是一个"bug"，NumPy中应该有什么东西需要修正才对。当你用整数数组选取大数组的子集时，最好还是注意一下这个问题：

In [76]: arr = randn(1000, 50)# 500行随机样本In [77]: inds = np.random.permutation(1000)[:500]In [78]: %timeit arr[inds]1000 loops, best of 3: 356 us per loopIn [79]: %timeit arr.take(inds, axis=0)10000 loops, best of 3: 34 us per loop

译注4

：实在找不到更好的词了，所以这里还是应该解释一下。虽然都是“重复”，但可以这样理解：duplicate是结果，replicate是造成duplicate的过程。

广播

广播（broadcasting）指的是不同形状的数组之间的算术运算的执行方式。它是一种非常强大的功能，但也容易令人误解，即使是经验丰富的老手也是如此。将标量值跟数组合并时就会发生最简的广播：

In [80]: arr = np.arange(5)In [81]: arr In [82]: arr * 4Out[81]: array([0, 1, 2, 3, 4]) Out[82]: array([0, 4, 8, 12, 16])

这里我们说：在这个乘法运算中，标量值4被广播到了其他所有的元素上。

再来看一个例子，我们可以通过减去列平均值的方式对数组的每一列进行距平化处理。这个问题解决起来非常简单：

In [83]: arr = randn(4, 3)In [84]: arr.mean(0)Out[84]: array([0.1321, 0.552 , 0.8571])In [85]: demeaned = arr - arr.mean(0)In [86]: demeaned In [87]: demeaned.mean(0)Out[86]: Out[87]: array([0., -0., -0.])array([[0.1718, -0.1972, -1.3669], [-0.1292, 1.6529, -0.3429], [-0.2891, -0.0435, 1.2322], [0.2465, -1.4122, 0.4776]])

图12-4形象地展示了该过程。用广播的方式对行进行距平化处理会稍微麻烦一些。幸运的是，只要遵循一定的规则，低维度的值是可以被广播到数组的任意维度的（比如对二维数组各列减去行平均值）。于是就得到了：

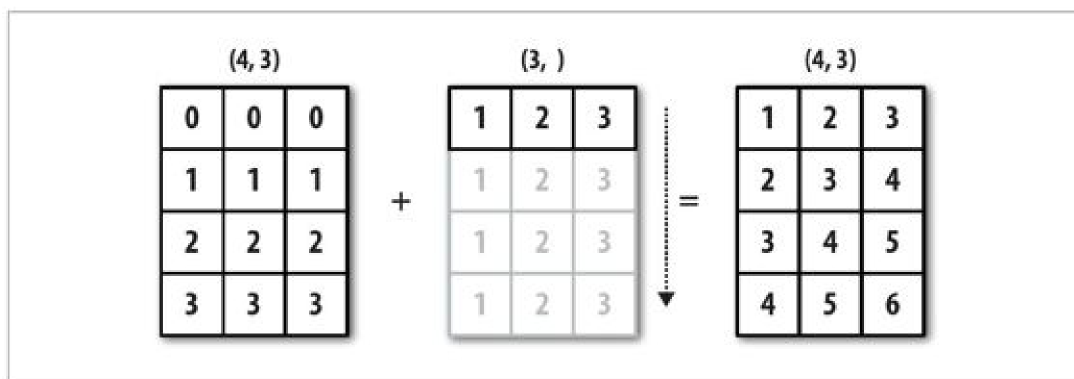


图12-4：一维数组在轴0上的广播

广播的原则

如果两个数组的后缘维度（trailing dimension，即从末尾开始算起的维度）的轴长度相符或其中一方的长度为1，则认为它们是广播兼容的。广播会在缺失和（或）长度为1的维度上进行。

虽然我是一名经验丰富的NumPy老手，但经常还是得停下来画张图并想想广播的原则。再来看一下最后那个例子，假设你希望对各行减去那个平均值。由于arr.mean(0)的长度为3，所以它可以在0轴向上进行广播：因为arr的后缘维度是3，所以它们是兼容的。根据该原则，要在1轴向上做减法（即各行减去行平均值），较小的那个数组的形状必须是(4,1)：

In [88]: arrOut[88]:array([[0.3039, 0.3548, -0.5097], [0.0029, 2.2049, 0.5142], [-0.1571, 0.5085, 2.0893], [0.3786, -0.8602, 1.3347]])In [89]: row_means = arr.mean(1) In [90]: row_means.reshape((4, 1)) Out[90]: array([[0.0496], [0.9073], [0.8136], [0.2844]])In [91]: demeaned = arr - row_means.reshape((4, 1))In [92]: demeaned.mean(1)Out[92]: array([0., 0., 0., 0.])

你的头还没炸吧？图12-5说明了该运算的过程。

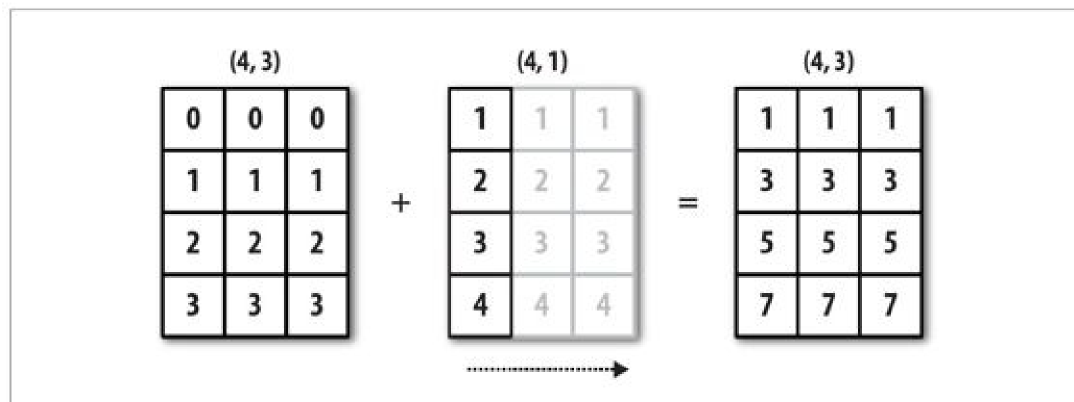


图12-5：二维数组在轴1上的广播

图12-6展示了另外一种情况，这次是在一个三维数组上沿0轴向加上一个二维数组。

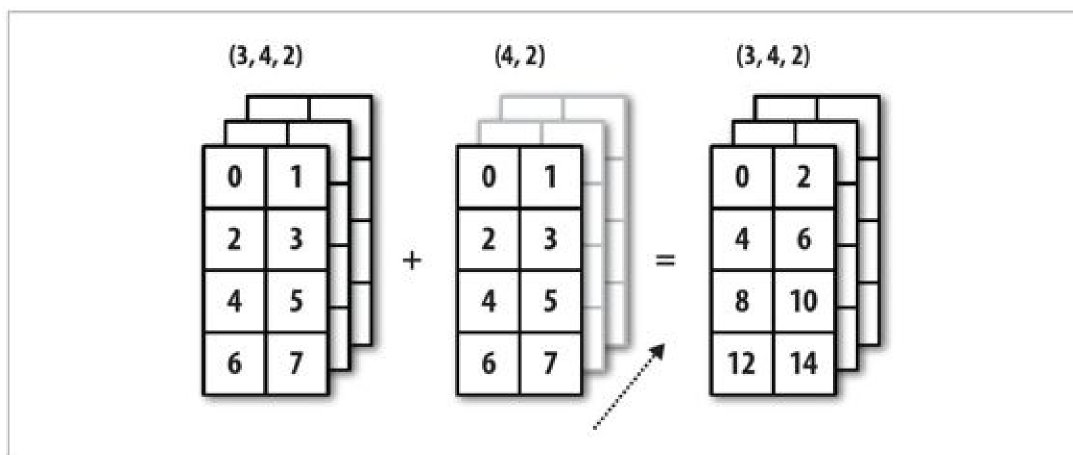


图12-6：三维数组在轴0上的广播

沿其他轴向广播

高维度数组的广播似乎更难以理解，而实际上它也是遵循广播原则的。如果不然，你就会得到下面这样一个错误：

```
In [93]: arr - arr.mean(1)---------------------------------------------------------------------------ValueError Traceback (most recent call last) in <module>()----> 1 arr - arr.mean(1)ValueError: operands could not be broadcast together with shapes (4,3) (4)
```

人们经常需要通过算术运算过程将较低维度的数组在除0轴以外的其他轴向上广播。根据广播的原则，较小数组的“广播维”必须为1。在上面那个行距平化的例子中，这就意味着要将行平均值的形状变成(4,1)而不是(4,)：

```
In [94]: arr - arr.mean(1).reshape((4, 1))Out[94]:array([[ 0.2542, 0.3051, -0.5594], [ -0.9044, 1.2976, -0.3931], [ -0.9707, -0.3051, 1.2757], [ 0.0942, -1.1446, 1.0503]])
```

对于三维的情况，在三维中的任何一维上广播其实也就是将数据重塑为兼容的形状而已。图12-7说明了要在三维数组各维度上广播的形状需求。

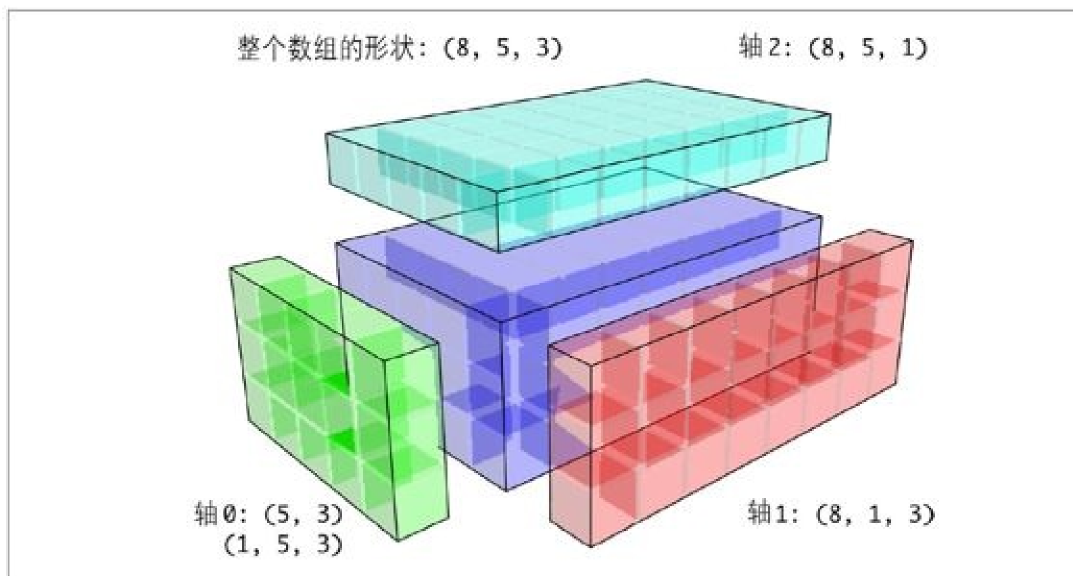


图12-7：能在该三维数组上广播的二维数组的形状

于是就有了一个非常普遍的问题（尤其是在通用算法中），即专门为了广播而添加一个长度为1的新轴。虽然reshape是一个办法，但插入轴需要构造一个表示新形状的元组。这是一个很郁闷的过程。因此，NumPy数组提供了一种通过索引机制插入轴的特殊语法。下面这段代码通过特殊的np.newaxis属性以及“全”切片来插入新轴：

In [95]: arr = np.zeros((4, 4))In [96]: arr_3d = arr[:, np.newaxis, :]In [97]: arr_3d.shapeOut[97]: (4, 1, 4)In [98]: arr_1d = np.random.normal(size=3)In [99]: arr_1d[:, np.newaxis] In [100]: arr_1d[np.newaxis, :]Out[99]: Out[100]: array([[-0.3899, 0.396 , -0.1852]])array([[-0.3899], [0.396], [-0.1852]])

因此，如果我们有一个三维数组，并希望对轴2进行距平化，那么只需要编写下面这样的代码就可以了：

In [101]: arr = randn(3, 4, 5)In [102]: depth_means = arr.mean(2)In [103]: depth_meansOut[103]:array([[0.1097, 0.3118, -0.5473, 0.2663], [0.1747, 0.1379, 0.1146, -0.4224], [0.0217, 0.3686, -0.0468, 1.3026]])In [104]: demeaned = arr - depth_means[:, :, np.newaxis]In [105]: demeaned.mean(2)Out[105]:array([[0., 0., -0., 0.], [0., -0., -0., 0.], [-0., -0., 0., 0.]])

也许你会对此感到非常困惑。不用担心，只要多动手，很快就能搞明白！

有些读者可能会想，在对指定轴进行距平化时，有没有一种既通用又不牺牲性能的方法呢？实际上是有的，但需要一些索引方面的技巧：

def demean_axis(arr, axis=0): means = arr.mean(axis) # 下面这些一般化的东西类似于N维的[:, :, np.newaxis] indexer = [slice(None)] * arr.ndim indexer[axis] = np.newaxis return arr - means[indexer]

通过广播设置数组的值

算术运算所遵循的广播原则同样也适用于通过索引机制设置数组值的操作。对于最简单的情况，我们可以这样做：

In [106]: arr = np.zeros((4, 3))In [107]: arr[:] = 5 In [108]: arr Out[108]: array([[5., 5., 5.], [5., 5., 5.], [5., 5., 5.], [5., 5., 5.]])

再看一个复杂点的例子，假设我们想要用一个一维数组来设置目标数组的各列。只要保证形状兼容就可以了：

In [109]: col = np.array([1.28, -0.42, 0.44, 1.6])In [110]: arr[:] = col[:, np.newaxis] In [111]: arr Out[111]: array([[1.28, 1.28, 1.28], [-0.42, -0.42, -0.42], [0.44, 0.44, 0.44], [1.6 , 1.6 , 1.6]])In [112]: arr[:2] = [[-1.37], [0.509]] In [113]: arr Out[113]: array([[-1.37 , -1.37 , -1.37], [0.509, 0.509, 0.509], [0.44 , 0.44 , 0.44], [1.6 , 1.6 , 1.6]])

ufunc高级应用

虽然许多NumPy用户只会用到通用函数所提供的快速的元素级运算，但通用函数实际上还有一些高级用法能使我们丢开循环而编写出更为简洁的代码。

ufunc实例方法

NumPy的各个二元ufunc都有一些用于执行特定矢量化运算的特殊方法。表12-2汇总了这些方法，下面我将通过几个具体的例子对它们进行说明。

reduce接受一个数组参数，并通过一系列的二元运算对其值进行聚合（可指明轴向）。例如，我们可以用np.add.reduce对数组中各个元素进行求和：

In [114]: arr = np.arange(10)In [115]: np.add.reduce(arr)Out[115]: 45In [116]: arr.sum()Out[116]: 45

起始值取决于ufunc（对于add的情况，就是0）。如果设置了轴号，约简运算就会沿该轴向执行。这就使你能用一种比较简洁的方式得到某些问题的答案。在下面这个例子中，我们用np.logical_and检查数组各行中的值是否是有序的：

In [118]: arr = randn(5, 5)In [119]: arr[::2].sort(1) # 对部分行进行排序In [120]: arr[:, :-1] < arr[:, 1:]Out[120]:array([[True, True, True, True], [False, True, False, False], [True, True, True, True], [True, False, True, True], [True, True, True, True]], dtype=bool)In [121]: np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=1)Out[121]: array([True, False, True, False, True], dtype=bool)

当然了，logical_and.reduce跟all方法是等价的。

accumulate跟reduce的关系就像cumsum跟sum的关系那样。它产生一个跟原数组大小相同的中间“累计”值数组：

In [122]: arr = np.arange(15).reshape((3, 5))In [123]: np.add.accumulate(arr, axis=1)Out[123]:array([[0, 1, 3, 6, 10], [5, 11, 18, 26, 35], [10, 21, 33, 46, 60]])

outer用于计算两个数组的叉积：

In [124]: arr = np.arange(3).repeat([1, 2, 2])In [125]: arrOut[125]: array([0, 1, 1, 2, 2])In [126]: np.multiply.outer(arr, np.arange(5))Out[126]:array([[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8], [0, 2, 4, 6, 8]])

outer输出结果的维度是两个输入数据的维度之和：

In [127]: result = np.subtract.outer(randn(3, 4), randn(5))In [128]: result.shapeOut[128]: (3, 4, 5)

最后一个方法reduceat用于计算“局部约简”，其实就是一个对数据各切片进行聚合的groupby运算。虽然其灵活性不如pandas的groupby功能，但它在适当的情况下运算会非常快。它接受一组用于指示如何对值进行拆分和聚合的“面元边界”：

In [129]: arr = np.arange(10)In [130]: np.add.reduceat(arr, [0, 5, 8])Out[130]: array([10, 18, 17])

最终结果是在arr[0:5]、arr[5:8]以及arr[8:]上执行的约简（本例中就是求和）。跟其他方法一样，这里也可以传入一个axis参数：

In [131]: arr = np.multiply.outer(np.arange(4), np.arange(5))In [132]: arr In [133]: np.add.reduceat(arr, [0, 2, 4], axis=1)Out[132]: Out[133]:array([[0, 0, 0, 0, 0], array([[0, 0, 0], [0, 1, 2, 3, 4], [1, 5, 4], [0, 2, 4, 6, 8], [2, 10, 8], [0, 3, 6, 9, 12]]) [3, 15, 12]])

表12-2：ufunc的方法

方法	说明
reduce(x)	通过连续执行原始运算的方式对值进行聚合
accumulate(x)	聚合值，保留所有局部聚合结果
reduceat(x, bins)	“局部”约简（也就是groupby）。约简数据的各个切片以产生聚合型数组
outer(x, y)	对x和y中的每对元素应用原始运算。结果数组的形状为x.shape + y.shape

自定义ufunc

有两个工具可以让你将自定义函数像ufunc那样使用。numpy.frompyfunc接受一个Python函数以及两个分别表示输入输出参数数量的整数。例如，下面是一个能够实现元素级加法的简单函数：

In [134]: def add_elements(x, y): .....: return x + yIn [135]: add_them = np.frompyfunc(add_elements, 2, 1)In [136]: add_them(np.arange(8), np.arange(8))Out[136]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)

用frompyfunc创建的函数总是返回Python对象数组，这一点很不方便。幸运的是，还有另一个办法，即numpy.vectorize。虽然没有frompyfunc那么强大，但它在类型推断方面要更智能一些：

In [137]: add_them = np.vectorize(add_elements, otypes=[np.float64])In [138]: add_them(np.arange(8), np.arange(8))Out[138]: array([0., 2., 4., 6., 8., 10., 12., 14.])

虽然这两个函数提供了一种创建ufunc型函数的手段，但它们非常慢，因为它们在计算每个元素时都要执行一次Python函数调用，这自然会比NumPy自带的基于C的ufunc慢很多：

In [139]: arr = randn(10000)In [140]: %timeit add_them(arr, arr)100 loops, best of 3: 2.12 ms per loopIn [141]: %timeit np.add(arr, arr)100000 loops, best of 3: 11.6 us per loop

为此，Python科学计算社区正在开发一些项目，力求使自定义ufunc的性能接近内置的那些。

结构化和记录式数组

你可能已经注意到了，到目前为止我们所讨论的ndarray都是一种同质数据容器，也就是说，在它所表示的内存块中，各元素占用的字节数相同（具体根据dtype而定）。从表面上，它似乎不能用于表示异质或表格型的数据。结构化数组是一种特殊的ndarray，其中的各个元素可以被看做C语言中的结构体（struct，这就是“结构化”的由来）或SQL表中带有多个命名字段的行：

In [142]: dtype = [('x', np.float64), ('y', np.int32)]In [143]: sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)In [144]: sarrOut[144]:array([(1.5, 6), (3.141592653589793, -2)], dtype=[('x', '<f8'), ('y', '<i4')])

定义结构化dtype（请参考NumPy的在线文档）的方式有很多。最典型的办法是元组列表，各元组的格式为(field_name,field_data_type)。这样，数组的元素就成了元组式的对象，该对象中各个元素可以像字典那样进行访问：

In [145]: sarr[0]Out[145]: (1.5, 6)In [146]: sarr[0]['y']Out[146]: 6

字段名保存在dtype.names属性中。在访问结构化数组的某个字段时，返回的是该数据的视图，所以不会发生数据复制：

In [147]: sarr['x']Out[147]: array([1.5 , 3.1416])

嵌套dtype和多维字段

在定义结构化dtype时，你可以再设置一个形状（可以是一个整数，也可以是一个元组）：

In [148]: dtype = [('x', np.int64, 3), ('y', np.int32)]In [149]: arr = np.zeros(4, dtype=dtype)In [150]: arrOut[150]:array([([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0)], dtype=[('x', '<i8', (3,)), ('y', '<i4')])

在这种情况下，各个记录的x字段所表示的是一个长度为3的数组：

In [151]: arr[0]['x']Out[151]: array([0, 0, 0])

访问arr['x']即可得到一个二维数组，而不是前面那个例子中的一维数组：

In [152]: arr['x']Out[152]:array([[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]])

这就使我们能用单个数组的内存块存放复杂的嵌套结构。既然dtype可以想怎么复杂就怎么复杂，那为什么不试试嵌套dtype呢？下面是一个简单的例子：

In [153]: dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y', np.int32)]In [154]: data = np.array([((1, 2), 5), ((3, 4), 6)], dtype=dtype)In [155]: data['x']Out[155]:array([(1.0, 2.0), (3.0, 4.0)], dtype=[('a', '<f8'), ('b', '<f4')])In [156]: data['y']Out[156]: array([5, 6], dtype=int32)In [157]: data['x']['a']Out[157]: array([1., 3.])

不难看出，可变形状的字段和嵌套记录是一种非常强大的功能。与此相比，pandas的DataFrame并不直接支持该功能，但它的分层索引机制跟这个差不多。

为什么要用结构化数组

跟pandas的DataFrame相比，NumPy的结构化数组是一种相对较低级的工具。它可以将单个内存块解释为带有任意复杂嵌套列的表格型结构。由于数组中的每个元素在内存中都被表示为固定的字节数，所以结构化数组能够提供非常快速高效的磁盘数据读写（包括内存映像，稍后将详细介绍）、网络传输等功能。

结构化数组的另一个常见用法是，将数据文件写成定长记录字节流，这是C和C++代码中常见的数据序列化手段（业界许多历史系统中都能找得到）。只要知道文件的格式（记录的大小、元素的顺序、字节数以及数据类型等），就可以用np.fromfile将数据读入内存。这种用法超出了本书的范围，只要知道有这么一回事就可以了。

结构化数组操作：numpy.lib.recfunctions

适用于结构化数组的函数没有DataFrame那么多。NumPy模块numpy.lib.recfunctions中有一些用于增删字段或执行基本连接运算的工具。对于这些工具，我们需要记住的是：一般都需要创建一个新数组以便对dtype进行修改（比如添加或删除一列）。这些函数就留给有兴趣的读者自己去研究了，因为本书中不会用到它们。

更多有关排序的话题

跟Python内置的列表一样，ndarray的sort实例方法也是就地排序。也就是说，数组内容的重新排列是不会产生新数组的：

In [158]: arr = randn(6)In [159]: arr.sort()In [160]: arrOut[160]: array([-1.082 , 0.3759, 0.8014, 1.1397, 1.2888, 1.8413])

在对数组进行就地排序时要注意一点：如果目标数组只是一个视图，则原始数组将会被修改：

In [161]: arr = randn(3, 5)In [162]: arrOut[162]:array([[-0.3318, -1.4711, 0.8705, -0.0847, -1.1329], [-1.0111, -0.3436, 2.1714, 0.1234, -0.0189], [0.1773, 0.7424, 0.8548, 1.038 , -0.329]])In [163]: arr[:, 0].sort() # Sort first column values in-placeIn [164]: arrOut[164]:array([[-1.0111, -1.4711, 0.8705, -0.0847, -1.1329], [-0.3318, -0.3436, 2.1714, 0.1234, -0.0189], [0.1773, 0.7424, 0.8548, 1.038 , -0.329]])

相反，numpy.sort会为原数组创建一个已排序副本。它所接受的参数（比如kind，稍后介绍）跟ndarray.sort一样：

In [165]: arr = randn(5)In [166]: arrOut[166]: array([-1.1181, -0.2415, -2.0051, 0.7379, -1.0614])In [167]: np.sort(arr)Out[167]: array([-2.0051, -1.1181, -1.0614, -0.2415, 0.7379])In [168]: arrOut[168]: array([-1.1181, -0.2415, -2.0051, 0.7379, -1.0614])

这两个排序方法都可以接受一个axis参数，以便沿指定轴向对各块数据进行单独排序：

In [169]: arr = randn(3, 5)In [170]: arrOut[170]:array([[0.5955, -0.2682, 1.3389, -0.1872, 0.9111], [-0.3215, 1.0054, -0.5168, 1.1925, -0.1989], [0.3969, -1.7638, 0.6071, -0.2222, -0.2171]])In [171]: arr.sort(axis=1)In [172]: arrOut[172]:array([[-0.2682, -0.1872, 0.5955, 0.9111, 1.3389], [-0.5168, -0.3215, -0.1989, 1.0054, 1.1925], [-1.7638, -0.2222, -0.2171, 0.3969, 0.6071]])

你可能注意到了，这两个排序方法都不可以被设置为降序。其实这也无所谓，因为数组切片会产生视图（也就是说，不会产生副本，也不需要任何其他的计算工作）。许多Python用户都很熟悉一个有关列表的小技巧：values[::-1]可以返回一个反序的列表。对ndarray也是如此：

In [173]: arr[:, ::-1]Out[173]:array([[1.3389, 0.9111, 0.5955, -0.1872, -0.2682], [1.1925, 1.0054, -0.1989, -0.3215, -0.5168], [0.6071, 0.3969, -0.2171, -0.2222, -1.7638]])

间接排序：argsort和lexsort

在数据分析工作中，常常需要根据一个或多个键对数据集进行排序。例如，一个有关学生信息的数据表可能需要以姓和名进行排序（先姓后名）。这就是间接排序的一个例子，如果你阅读过有关pandas的章节，那就已经见过不少高级例子了。给定一个或多个键，你就可以得到一个由整数组成的索引数组（我亲切地称之为索引器），其中的索引值说明了数据在新顺序下的位置。argsort和numpy.lexsort就是实现该功能的两个主要方法。下面是一个简单的例子：

In [174]: values = np.array([5, 0, 1, 3, 2])In [175]: indexer = values.argsort()In [176]: indexerOut[176]: array([1, 2, 4, 3, 0])In [177]: values[indexer]Out[177]: array([0, 1, 2, 3, 5])

下面这段代码根据数组的第一行对其进行排序：

In [178]: arr = randn(3, 5)In [179]: arr[0] = valuesIn [180]: arrOut[180]:array([[5. , 0. , 1. , 3. , 2.], [-0.3636, -0.1378, 2.1777, -0.4728, 0.8356], [-0.2089, 0.2316, 0.728 , -1.3918, 1.9956]])In [181]: arr[:, arr[0].argsort()]Out[181]:array([[0. , 1. , 2. , 3. , 5.], [-0.1378, 2.1777, 0.8356, -0.4728, -0.3636], [0.2316, 0.728 , 1.9956, -1.3918, -0.2089]])

lexsort跟argsort差不多，只不过它可以一次性对多个键数组执行间接排序（字典序）。假设我们想对一些以姓和名标识的数据进行排序：

In [182]: first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill', 'Barbara'])In [183]: last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones', 'Walters'])In [184]: sorter = np.lexsort((first_name, last_name))In [185]: zip(last_name[sorter], first_name[sorter])Out[185]:[('Arnold', 'Jane'), ('Arnold', 'Steve'), ('Jones', 'Bill'), ('Jones', 'Bob'), ('Walters', 'Barbara')]

刚开始使用lexsort的时候可能会比较容易头晕，这是因为键的应用顺序是从最后一个传入的算起的。不难看出，last_name是先于first_name被应用的。

注意： Series和DataFrame的sort_index以及Series的order方法就是通过这些函数的变体（它们还必须考虑缺失值）实现的。

其他排序算法

稳定的（stable）排序算法会保持等价元素的相对位置。对于相对位置具有实际意义的那些间接排序而言，这一点非常重要：

In [186]: values = np.array(['2:first', '2:second', '1:first', '1:second', '1:third'])In [187]: key = np.array([2, 2, 1, 1, 1])In [188]: indexer = key.argsort(kind='mergesort')In [189]: indexerOut[189]: array([2, 3, 4, 0, 1])In [190]: values.take(indexer)Out[190]:array(['1:first', '1:second', '1:third', '2:first', '2:second'], dtype='|S8')

mergesort（合并排序）是唯一的稳定排序

译注5

，它保证有O(n log n)的性能（空间复杂度），但是其平均性能比默认的quicksort（快速排序）要差。表12-3列出了可用的排序算法及其相关的性能指标。大部分用户完全不需要知道这些东西，但了解一下总是好的。

表12-3：数组排序算法

kind	速度	稳定性	工作空间	最坏的情况
'quicksort'	1	否	0	$O(n^2)$
'mergesort'	2	是	n/2	O(n log n)
'heapsort'	3	否	0	O(n log n)

警告： 到编写本书时为止，Python对象（dtype=object）数组可用的排序算法只有quicksort。也就是说，在处理Python对象时如果需要用到稳定排序，那就得自己想办法了。

numpy.searchsorted：在有序数组中查找元素

searchsorted是一个在有序数组上执行二分查找的数组方法，只要将值插入到它返回的那个位置就能维持数组的有序性：

In [191]: arr = np.array([0, 1, 7, 12, 15])In [192]: arr.searchsorted(9)Out[192]: 3

你可能已经想到了，传入一组值就能得到一组索引：

In [193]: arr.searchsorted([0, 8, 11, 16])Out[193]: array([0, 3, 3, 5])

从上面的结果中可以看出，对于元素0，searchsorted会返回0。这是因为其默认行为是返回相等值组的左侧索引：

In [194]: arr = np.array([0, 0, 0, 1, 1, 1, 1])In [195]: arr.searchsorted([0, 1])Out[195]: array([0, 3])In [196]: arr.searchsorted([0, 1], side='right')Out[196]: array([3, 7])

再来看searchsorted的另一个用法，假设我们有一个数据数组（其中的值在0到10000之间），还有一个表示“面元边界”的数组，我们希望用它将数据数组拆分开：

In [197]: data = np.floor(np.random.uniform(0, 10000, size=50))In [198]: bins = np.array([0, 100, 1000, 5000, 10000])In [199]: dataOut[199]:array([8304., 4181., 9352., 4907., 3250., 8546., 2673., 6152., 2774., 5130., 9553., 4997., 1794., 9688., 426., 1612., 651., 8653., 1695., 4764., 1052., 4836., 8020., 3479., 1513., 5872., 8992., 7656., 4764., 5383., 2319., 4280., 4150., 8601., 3946., 9904., 7286., 9969., 6032., 4574., 8480., 4298., 2708., 7358., 6439., 7916., 3899., 9182., 871., 7973.])

然后，为了得到各数据点所属区间的编号（其中1表示面元[0,100)），我们可以直接使用searchsorted：

In [200]: labels = bins.searchsorted(data)In [201]: labelsOut[201]:array([4, 3, 4, 3, 3, 4, 3, 4, 3, 4, 4, 3, 3, 4, 2, 3, 2, 4, 3, 3, 3, 3, 4, 3, 3, 4, 4, 4, 3, 4, 3, 3, 3, 4, 3, 4, 4, 4, 4, 3, 4, 3, 3, 4, 4, 4, 3, 4, 2, 4])

通过pandas的groupby使用该结果即可非常轻松地对原数据集进行拆分：

In [202]: Series(data).groupby(labels).mean()Out[202]:2 649.3333333 3411.5217394 7935.041667

注意，其实NumPy的digitize函数也可用于计算这种面元编号：

In [203]: np.digitize(data, bins)Out[203]:array([4, 3, 4, 3, 3, 4, 3, 4, 3, 4, 4, 3, 3, 4, 2, 3, 2, 4, 3, 3, 3, 3, 4, 3, 3, 4, 4, 4, 3, 4, 3, 3, 3, 4, 3, 4, 4, 4, 4, 3, 4, 3, 3, 4, 4, 4, 3, 4, 2, 4])

译注5：只是这三种里面唯一稳定的而已。

NumPy的matrix类

跟其他面向矩阵算和线性代数的语言相比（如MATLAB、GAUSS等），NumPy的线性代数语法往往比较繁琐。其中一个原因是，矩阵操作需要用到numpy.dot。再加上NumPy的索引语义也不同，所以有时不那么容易将代码移植到Python译注6。从二维数组中选取一行（比如X[1,:]）或一列（如X[:,1]）将会产生一个一维数组，而在MATLAB中则是二维数组。

In [204]: X = np.array([[8.82768214, 3.82222409, -1.14276475, 2.04411587], ...: [3.82222409, 6.75272284, 0.83909108, 2.08293758], ...: [-1.14276475, 0.83909108, 5.01690521, 0.79573241], ...: [2.04411587, 2.08293758, 0.79573241, 6.24095859]])In [205]: X[:, 0] # 一维的Out[205]: array([8.8277, 3.8222, -1.1428, 2.0441])In [206]: y = X[:, :1] # 切片操作可产生二维结果In [207]: XOut[207]:array([[8.8277, 3.8222, -1.1428, 2.0441], [3.8222, 6.7527, 0.8391, 2.0829], [-1.1428, 0.8391, 5.0169, 0.7957], [2.0441, 2.0829, 0.7957, 6.241]])In [208]: yOut[208]:array([[8.8277], [3.8222], [-1.1428], [2.0441]])

在这个问题中，积yT×y会被表达成下面这个样子：

In [209]: np.dot(y.T, np.dot(X, y))Out[209]: array([[1195.468]])

为了不用编写大量的矩阵运算代码，NumPy提供了一个matrix类，其索引行为更像MATLAB：单行或列会以二维形式返回，且使用星号（*）的乘法直接就是矩阵乘法。上面那些运算用numpy.matrix来编写的话，应该是下面这个样子：

In [210]: Xm = np.matrix(X)In [211]: ym = Xm[:, 0]In [212]: XmOut[212]:matrix([[8.8277, 3.8222, -1.1428, 2.0441], [3.8222, 6.7527, 0.8391, 2.0829], [-1.1428, 0.8391, 5.0169, 0.7957], [2.0441, 2.0829, 0.7957, 6.241]])In [213]: ymOut[213]:matrix([[8.8277], [3.8222], [-1.1428], [2.0441]])In [214]: ym.T *Xm* ymOut[214]: matrix([[1195.468]])

matrix还有一个特殊的属性I，其功能是返回矩阵的逆：

In [215]: Xm.I * XOut[215]:matrix([[1., -0., -0., -0.], [0., 1., 0., 0.], [0., 0., 1., 0.], [0., 0., 0., 1.]])

我不建议用numpy.matrix替代正规的ndarray，因为它们的应用面较窄。对于个别带有大量线性代数运算的函数，可以将函数参数转换为matrix类型，然后在返回之前用np.asarray（不会复制任何数据）将其转换回正规的ndarray。

译注6

：原文有歧义，根据上下文的意思，应该是说不容易把其他语言的代码移植过来。

高级数组输入输出

我在第4章中讲过，np.save和np.load可用于读写磁盘上以二进制格式存储的数组。其实还有一些工具可用于更为复杂的场景。尤其是内存映像（memory map），它使你能处理在内存中放下的数据集。

内存映像文件

内存映像文件是一种将磁盘上的非常大的二进制数据文件当做内存中的数组进行处理的方式。NumPy实现了一个类似于ndarray的memmap对象，它允许将大文件分成小段进行读写，而不是一次性将整个数组读入内存。memmap也拥有跟普通数组一样的方法，因此，基本上只要是能用于ndarray的算法就也能用于memmap。

使用函数np.memmap并传入一个文件路径、数据类型、形状以及文件模式，即可创建一个新的memmap：

In [216]: mmap = np.memmap('mymmap', dtype='float64', mode='w+', shape=(10000, 10000))In [217]: mmapOut[217]:memmap([[0., 0., 0., ..., 0., 0., 0.], [0., 0., 0., ..., 0., 0., 0.], [0., 0., 0., ..., 0., 0., 0.], ..., [0., 0., 0., ..., 0., 0., 0.], [0., 0., 0., ..., 0., 0., 0.], [0., 0., 0., ..., 0., 0., 0.]])

对memmap切片将会返回磁盘上的数据的视图：

In [218]: section = mmap[:5]

如果将数据赋值给这些视图：数据会先被缓存在内存中（就像是Python的文件对象），调用flush即可将其写入磁盘。

In [219]: section[:] = np.random.randn(5, 10000)In [220]: mmap.flush()In [221]: mmapOut[221]:memmap([[-0.1614, -0.1768, 0.422 , ..., -0.2195, -0.1256, -0.4012], [0.4898, -2.2219, -0.7684, ..., -2.3517, -1.0782, 1.3208], [-0.6875, 1.6901, -0.7444, ..., -1.4218, -0.0509, 1.2224], ..., [0. , 0. , 0. , ..., 0. , 0. , 0.], [0. , 0. , 0. , ..., 0. , 0. , 0.], [0. , 0. , 0. , ..., 0. , 0. , 0.]])In [222]: del mmap

只要某个内存映像超出了作用域，它就会被垃圾回收器回收，之前对其所做的任何修改都会被写入磁盘。当打开一个已经存在的内存映像时，仍然需要指明数据类型和形状，因为磁盘上的那个文件只是一块二进制数据而已，没有任何元数据：

In [223]: mmap = np.memmap('mymmap', dtype='float64', shape=(10000, 10000))In [224]: mmapOut[224]:memmap([[-0.1614, -0.1768, 0.422 , ..., -0.2195, -0.1256, -0.4012], [0.4898, -2.2219, -0.7684, ..., -2.3517, -1.0782, 1.3208], [-0.6875, 1.6901, -0.7444, ..., -1.4218, -0.0509, 1.2224], ..., [0. , 0. , 0. , ..., 0. , 0. , 0.], [0. , 0. , 0. , ..., 0. , 0. , 0.], [0. , 0. , 0. , ..., 0. , 0. , 0.]])

由于内存映像其实就是一个存放在磁盘上的ndarray，所以完全可以使用前面介绍的结构化dtype。

HDF5及其他数组存储方式

PyTables和h5py这两个Python项目可以将NumPy的数组数据存储为高效且可压缩的HDF5格式（HDF意思是“层次化数据格式”）。你可以安全地将好几百GB甚至TB的数据存储为HDF5格式。很遗憾，这些库的用法超出了本书的范围。

PyTables提供了一些用于结构化数组的高级查询功能，而且还能添加列索引以提升查询速度。这跟关系型数据库所提供的表索引功能非常类似。

性能建议

使用NumPy的代码的性能一般都很不错，因为数组运算一般都比纯Python循环快得多。下面大致列出了一些需要注意的事项：

·将Python循环和条件逻辑转换为数组运算和布尔数组运算。

·尽量使用广播。

·避免复制数据，尽量使用数组视图（即切片）。

·利用ufunc及其各种方法。

如果单用NumPy无论如何都达不到所需的性能指标，就可以考虑一下用C、Fortran或Cython（等下会稍微介绍一下）来编写代码。我自己在工作中经常会用到Cython（http://cython.org），因为它不用花费我太多精力就能得到C语言那样的性能。

连续内存的重要性

虽然这个话题有点超出本书的范围，但还是要提一下，因为在某些应用场景中，数组的内存布局可以对计算速度造成极大的影响。这是因为性能差别在一定程度上跟CPU的高速缓存（cache）体系有关。运算过程中访问连续内存块（例如，对以C顺序存储的数组的行求和）一般是最快的，因为内存子系统会将适当的内存块缓存到超高速的L1或L2CPU Cache中

译注7

。此外，NumPy的C语言基础代码（某些）对连续存储的情况进行了优化处理，这样就能避免一些跨越式的内存访问。

一个数组的内存布局是连续的，就是说元素是以它们在数组中出现的顺序（即Fortran型（列优先）或C型（行优先））存储在内存中的。默认情况下，NumPy数组是以C型连续的方式创建的。列优先的数组（比如C型连续数组的转置）也被称为Fortran型连续。通过ndarray的flags属性即可查看这些信息：

In [227]: arr_c = np.ones((1000, 1000), order='C')In [228]: arr_f = np.ones((1000, 1000), order='F')In [229]: arr_c.flags In [230]: arr_f.flagsOut[229]: Out[230]: C_CONTIGUOUS : True C_CONTIGUOUS : False F_CONTIGUOUS : False F_CONTIGUOUS : True OWNDATA : True OWNDATA : True WRITEABLE : True WRITEABLE : True ALIGNED : True ALIGNED : True UPDATEIFCOPY : False UPDATEIFCOPY : FalseIn [231]: arr_f.flags.f_contiguousOut[231]: True

在这个例子中，对两个数组的行进行求和计算，理论上说，arr_c会比arr_f快，因为arr_c的行在内存中是连续的。我们可以在IPython中用%timeit来确认一下：

In [232]: %timeit arr_c.sum(1)1000 loops, best of 3: 1.33 ms per loopIn [233]: %timeit arr_f.sum(1)100 loops, best of 3: 8.75 ms per loop

如果想从NumPy中提升性能，这里就应该是下手的地方。如果数组的内存顺序不符合你的要求，使用copy并传入'C'或'F'即可解决该问题：

In [234]: arr_f.copy('C').flagsOut[234]: C_CONTIGUOUS : True F_CONTIGUOUS : False OWNDATA : True WRITEABLE : True ALIGNED : True UPDATEIFCOPY : False

注意，在构造数组的视图时，其结果不一定是连续的：

In [235]: arr_c[:50].flags.contiguous In [236]: arr_c[:, :50].flagsOut[235]: True Out[236]: C_CONTIGUOUS : False F_CONTIGUOUS : False OWNDATA : False WRITEABLE : True ALIGNED : True UPDATEIFCOPY : False

其他加速手段：Cython、f2py、C

近年来，Cython项目（http://cython.org）已经受到了许多Python程序员的认可，用它实现的代码运行速度很快（可能需要与C或C++库交互，但无需编写纯粹的C代码）。你可以将Cython看成是带有静态类型并能嵌入C函数的Python。下面这个简单的Cython函数用于对一个一维数组的所有元素求和：

from numpy cimport ndarray, float64_tdef sum_elements(ndarray[float64_t] arr): cdef Py_ssize_t i, n = len(arr) cdef float64_t result = 0 for i in range(n): result += arr[i] return result

Cython处理这段代码时，先将其翻译为C代码，然后编译这些C代码并创建一个Python扩展。Cython是一种诱人的高性能计算方式，因为编写Cython代码只比编写纯Python代码多花一点时间而已，而且还能跟NumPy紧密结合。一般的工作流程是：得到能在Python中运行的算法，然后再将其翻译为Cython（只需添加类型定义并完成一些其他必要的工作即可）。更多信息请参考该项目的文档。

其他有关NumPy的高性能代码编写手段还有f2py（FORTRAN 77和90的包装器生成器）以及利用纯C语言编写Python扩展。

译注7

：这里主要考虑的是预读机制以及缓存块失效率。由于这个存储层次是纯硬件实现的，谁的程序都控制不了，所以数据最好连续存储。

附录A　Python语言精要

知识是一座宝库，而实践就是开启这座宝库的钥匙。

——Thomas Fuller

人们常常问我要有关学习Python数据处理方面的优质资源。虽然市面上有许多非常不错的讲解Python语言的图书，但我在推荐的时候经常还是会犹豫不决，因为它们都是针对普通读者的，没有为那些只想"加载点儿数据，做点计算，再画点儿图"的读者做专门的裁剪。其实有几本书确实是关于Python科学计算编程的，但它们是专为数值计算和工程应用而设计的：解微分方程、计算积分、做蒙特卡罗模拟，以及其他各种数学方面的主题，但就是没有数据分析和统计方面的。由于本书的目的是让大家成为Python数据处理方面的熟手，所以我认为有必要花点时间从结构化和非结构化数据处理的角度重点介绍一些有关Python内置数据结构和库的最重要的功能。我将只介绍一些大致的信息，只要对本书的学习够用就行。

本附录并没有打算成为Python语言的详尽指南，只会对书中反复用到的那些功能做一个基本的概述。对于Python新手而言，我建议在读完本附录后再看看Python的官方教程（http://docs.python.org），最好能再读一两本有关Python通用编程方面的优质图书。以我的观点来看，如果只需要用Python进行高效的数据分析工作，根本就没必要非得成为通用软件编程方面的专家不可。我强烈建议你用IPython实验所有的代码示例，并查看各种类型、函数以及方法的文档。注意，这些例子中所用到的一些代码暂时还没必要解释得那么详细。

本书主要关注的是能够处理大数据集的高性能数组计算工具。为了使用这些工具，你常常得先把那些乱七八糟的数据处理成漂亮点的结构化形式。好在Python是一种最易上手的数据整形语言。你的Python语言能力越强，数据分析的准备工作就越简单。

Python解释器

Python是一种解释型语言。Python解释器是通过“一次执行一条语句”的方式运行程序的。标准的交互式Python解释器可以在命令行上通过python命令启动：

$ pythonPython 2.7.2 (default, Oct 4 2011, 20:06:09)[GCC 4.6.1] on linux2Type "help", "copyright", "credits" or "license" for more information.>>> a = 5>>> print a5

上面的">>>"是提示符，你可以在那里输入表达式。要退出Python解释器并返回命令提示符，输入exit()或按下Ctrl-D即可。

运行Python程序的方式很简单，只需调用python并将一个.py文件作为其第一个参数即可。假设我们已经创建了一个hello_world.py，其内容如下：

print 'Hello world'

只需在终端上输入如下命令即可运行：

$ python hello_world.pyHello world

虽然许多Python程序员用这种方式执行他们的所有Python代码，但Python科学计算程序员则更趋向于使用IPython（一种加强的交互式Python解释器）。第3章专门介绍了IPython系统。通过使用%run命令，IPython会在同一个进程中执行指定文件中的代码。因此，在这些代码执行完毕之后，你就可以通过交互的方式研究其结果了。

$ ipythonPython 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul 3 2011, 15:17:51)Type "copyright", "credits" or "license" for more information.IPython 0.12 -- An enhanced Interactive Python.? -> Introduction and overview of IPython's features.%quickref -> Quick reference.help -> Python's own help system.object? -> Details about 'object', use 'object??' for extra details.In [1]: %run hello_world.pyHello worldIn [2]:

默认的IPython提示符采用的是一种编号的风格（如In [2]:），而不是标准的">>>"提示符。

基础知识

语言语义

Python语言的设计特点是重视可读性、简洁性以及明确性。有些人甚至将它看做“可执行的伪码”。

缩进，而不是大括号

Python是通过空白符（制表符或空格）来组织代码的，不像其他语言（如R、C++、Java、Perl等）用的是大括号。以for循环为例，要实现前面说的那个快速排序算法：

for x in array: if x < pivot: less.append(x) else: greater.append(x)

冒号表示一段缩进代码块的开始，其后的所有代码都必须缩进相同的量，直到代码块结束为止。在别的语言中，你可能会看到下面这样的东西：

for x in array { if x < pivot { less.append(x) } else { greater.append(x) }}

使用空白符的主要好处是，它能使大部分Python代码在外观上看起来差不多。也就是说，当你阅读某段不是自己编写的（或一年前匆忙编写的）代码时不怎么容易出现“认知失调”。在那些空白符无实际意义的语言中，你可能会发现一些格式不统一的代码，比如：

for x in array { if x < pivot { less.append(x) } else { greater.append(x) } }

无论对它是爱是恨，反正有意义的空白符就是Python程序员的生活现实。再说了，以我的经验来看，它能使Python代码具有更高的可读性（至少比我用过其他语言要高）。虽然第一眼看上去会觉得比较火星，但我相信不用多久你就会喜欢上它的。

注意： 我强烈建议用4个空格作为默认缩进量，这样，你的编辑器就会将制表符替换为4个空格。许多文本编辑器都有一个这样的设置项。有些人喜欢用制表符或其他数量的空格，但用2个空格的情况非常少见。4个空格其实就是一种标准，绝大部分Python程序员都这么用。所以我建议：除非有特殊的原因，否则就用4个空格吧。

到目前为止，你可以看到，Python语句还能不以分号结束。不过分号还是可以用的，比如在一行上分隔多条语句：

a = 5; b = 6; c = 7

在一行上放置多条语句的做法在Python中一般是不推荐的，因为这往往会使代码的可读性变差。

万物皆对象

Python语言的一个重要特点就是其对象模型的一致性。Python解释器中的任何数值、字符串、数据结构、函数、类、模块等都待在它们自己的“盒子”里，而这个“盒子”也就是Python对象。每个对象都有一个与之相关的类型（比如字符串或函数）以及内部数据。在实际工作当中，这使得Python语言变得非常灵活，因为即使是函数也能被当做其他对象那样处理。

注释

任何前缀为井号（#）的文本都会被Python解释器忽略掉。这通常用于在代码中添加注释。有时你可能只是想排除不运行某些代码块而不想删除它们。最简单的办法就是注释掉那些代码：

results = []for line in file_handle: # 暂时保留空行 # if len(line) == 0: # continue results.append(line.replace('foo', 'bar'))

函数调用和对象方法调用

函数的调用需要用到圆括号以及0个或多个参数，此外还可以将返回值赋值给一个变量：

result = f(x, y, z)g()

几乎所有的Python对象都有一些附属函数（也就是方法），它们可以访问该对象的内部数据。方法的调用是这样写的：

obj.some_method(x, y, z)

函数既可以接受位置参数，也可以接受关键字参数：

result = f(a, b, c, d=5, e='foo')

稍后将详细介绍这个内容。

变量和按引用传递

在Python中对变量赋值时，你其实是在创建等号右侧对象的一个引用。用实际的例子来说吧，看看下面这个整数列表：

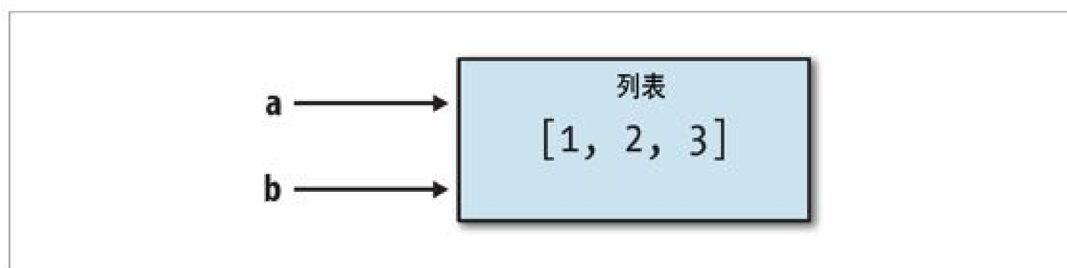
In [241]: a = [1, 2, 3]

假如我们将a赋值给一个新变量b：

In [242]: b = a

在某些语言中，该赋值过程将会导致数据[1,2,3]被复制。而在Python中，a和b现在都指向同一个对象，即原始列表[1,2,3]（如图A-1所示）。你可以自己验证一下：对a添加一个元素，然后看看b的情况。

In [243]: a.append(4)In [244]: bOut[244]: [1, 2, 3, 4]



图A-1：指向同一个对象的两个引用

理解Python引用的语义以及数据复制的条件、方式、原因等知识对于在Python中处理大数据集非常重要。

注意：赋值（assignment）操作也叫做绑定（binding），因为我们其实是将一个名称和一个对象绑定到一起。已经赋过值的变量名有时也被称为已绑定变量（bound variable）。

当你将对象以参数的形式传入函数时，其实只是传入了一个引用而已，不会发生任何复制。因此，Python被称为是按引用传递的，而某些其他的语言则既支持按值传递（创建副本）又支持按引用传递。也就是说，Python函数可以修改其参数的内容。假设我们有下面这样的一个函数：

def append_element(some_list, element): some_list.append(element)

根据刚才所说的，下面这样的结果应该是在意料之中的：

In [2]: data = [1, 2, 3]In [3]: append_element(data, 4)In [4]: dataOut[4]: [1, 2, 3, 4]

动态引用，强类型

跟许多编译型语言（如Java和C++）相反，Python中的对象引用没有与之关联的类型信息。下面这些代码不会有什么问题：

In [245]: a = 5In [246]: type(a)Out[246]: intIn [247]: a = 'foo'In [248]: type(a)Out[248]: str

变量其实就是对象在特定命名空间中的名称而已。对象的类型信息是保存在它自己内部的。有些人可能会轻率地认为Python不是一种“类型化语言”。其实不是这样的。看看下面这个例子：

In [249]: '5' + 5---TypeError Traceback (most recent call last) in <module>()----> 1 '5' + 5TypeError: cannot concatenate 'str' and 'int' objects

在有些语言中（比如Visual Basic），字符串'5'可能会被隐式地转换为整数，于是就会得到10。而在另一些语言中（比如JavaScript），整数5可能会被转换为字符串，于是就会得到'55'。而在这一点上，Python可以被认为是一种强类型语言，也就是说，所有对象都有一个特定的类型（或类），隐式转换只在很明显的情况下才会发生，比如下面这样：

In [250]: a = 4.5In [251]: b = 2# 这个操作是字符串格式化，稍后介绍In [252]: print 'a is %s, b is %s' % (type(a), type(b))a is , b is In [253]: a / bOut[253]: 2.25

了解对象的类型是很重要的。要想编写能够处理多个不同类型输入的函数就必须了解有关类型的知识。通过isinstance函数，你可以检查一个对象是否是某个特定类型的实例：

In [254]: a = 5In [255]: isinstance(a, int)Out[255]: True

isinstance可以接受由类型组成的元组。如果想检查某个对象的类型是否属于元组中所指定的那些：

In [256]: a = 5; b = 4.5In [257]: isinstance(a, (int, float))Out[257]: TrueIn [258]: isinstance(b, (int, float))Out[258]: True

属性和方法

Python中的对象通常都既有属性（attribute，即存储在该对象“内部”的其他Python对象）又有方法（method，与该对象有关的能够访问其内部数据的函数）。它们都能通过obj.attribute_name这样的语法进行访问：

In [1]: a = 'foo'In [2]: a.<Tab>a.capitalize a.format a.isupper a.rindex a.stripa.center a.index a.join a.rjust a.swapcasea.count a.isalnum a.ljust a.rpartition a.titlea.decode a.isalpha a.lower a.rsplit a.translatea.encode a.isdigit a.lstrip a.rstrip a.uppera.endswith a.islower a.partiti a.split a.zfilla.expandtabs a.isspace a.replace a.splitlinesa.find a.istitle a.rfind a.startswith

属性和方法还可以利用getattr函数通过名称进行访问：

> > > getattr(a, 'split')

虽然本书没怎么用到getattr函数以及与之相关的hasattr和setattr函数，但是它们还是很实用的，尤其是在编写通用的、可复用的代码时。

“鸭子”类型

译注1

一般来说，你可能不会关心对象的类型，而只是想知道它到底有没有某些方法或行为。比如说，只要一个对象实现了迭代器协议（iterator protocol），你就可以确认它是可迭代的。对于大部分对象而言，这就意味着它拥有一个**iter**魔术方法。当然，还有一个更好一些的验证办法，即尝试使用iter函数：

def isiterable(obj): try: iter(obj) return True except TypeError: # 不可迭代 return False

对于字符串以及大部分Python集合类型，该函数会返回True：

In [260]: isiterable('a string') In [261]: isiterable([1, 2, 3])Out[260]: True Out[261]: TrueIn [262]: isiterable(5)Out[262]: False

我常常在编写需要处理多类型输入的函数时用到这个功能。还有一种常见的应用场景：编写可以接受任何序列（列表、元组、ndarray）或迭代器的函数。你可以先检查对象是不是列表（或NumPy数组），如果不是，就将其转换成是：

if not isinstance(x, list) and isiterable(x): x = list(x)

引入（import）

在Python中，模块（module）就是一个含有函数和变量定义以及从其他.py文件引入的此类东西的.py文件。假设我们有下面这样一个模块：

some_module.pyPI = 3.14159def f(x):

return x + 2def g(a, b): return a + b

如果想要引入some_module.py中定义的变量和函数，我们可以在同一个目录下创建另一个文件：

import some_moduleresult = some_module.f(5)pi = some_module.PI

还可以写成这样：

from some_module import f, g, PIresult = g(5, PI)

通过as关键字，你可以引入不同的变量名译注2：

import some_module as smfrom some_module import PI as pi, g as gfr1 = sm.f(pi)r2 = gf(6, pi)

二元运算符和比较运算符

大部分二元数学运算和比较运算都跟我们想象中的一样：

In [263]: 5 - 7 In [264]: 12 + 21.5Out[263]: -2 Out[264]: 33.5In [265]: 5 <= 2Out[265]: False

表A-1中列出了所有可用的二元运算符。

要判断两个引用是否指向同一个对象，可以使用is关键字。如果想判断两个引用是否不是指向同一个对象，则可以使用is not：

In [266]: a = [1, 2, 3]In [267]: b = a# 注意，list函数始终会创建新列表In [268]: c = list(a)In [269]: a is bOut[269]: TrueIn [270]: a is not cOut[270]: True

注意，这跟比较运算"=="不是一回事，因为对于上面这个情况，我们将会得到：

In [271]: a == cOut[271]: True

is和is not常常用于判断变量是否为None，因为None的实例只有一个：

In [272]: a = NoneIn [273]: a is NoneOut[273]: True

表A-1：二元运算符

运算	说明
a + b	a加b
a - b	a减b
a * b	a乘以b
a / b	a除以b
a // b	a除以b后向下圆整，丢弃小数部分
a ** b	a的b次方
a & b	如果a和b均为True，则结果为True。对于整数，执行按位与操作
a | b	如果a和b任何一个为True，则结果为True。对于整数，执行按位或操作
a ^ b	对于布尔值，如果a或b为True（但不都为True），则结果为True。对于整数，执行按位异或操作
a == b	如果a等于b，则结果为True
a != b	如果a不等于b，则结果为True
a <= b、a < b	如果a小于等于（或小于）b，则结果为True
a > b、a >= b	如果a大于（或大于等于）b，则结果为True
a is b	如果引用a和b指向同一个Python对象，则结果为True
a is not b	如果引用a和b指向不同的Python对象，则结果为True

严格与懒惰

无论使用什么编程语言，都必须了解表达式是何时被求值的。看看下面这两个简单的表达式：

a = b = c = 5d = a + b * c

在Python中，只要这些语句被求值，相关计算就会立即（也就是严格）发生，d的值会被设置为30。而在另一种编程范式中（比如Haskell这样的纯函数编程语言），d的值在被使用之前是不会被计算出来的。这种将计算推迟的思想通常称为延迟计算（lazy evaluation

译注3

）。而Python是一种非常严格的（急性子）语言。几乎在任何时候，计算过程和表达式都是立即求值的。即使是在上面那个简单的例子中，也是先计算b *c的结果然后再将其与a加起来的。

有一些Python技术（尤其是用到迭代器和生成器的那些）可以用于实现延迟计算。在数据密集型应用中，当执行一些负荷非常高的计算时（这种情况不太多），这些技术就能派上用场了。

可变和不可变的对象

大部分Python对象是可变的（mutable），比如列表、字典、NumPy数组以及大部分用户自定义类型（类）。也就是说，它们所包含的对象或值是可以被修改的。

In [274]: a_list = ['foo', 2, [4, 5]]In [275]: a_list[2] = (3, 4)In [276]: a_listOut[276]: ['foo', 2, (3, 4)]

而其他的（如字符串和元组等）则是不可变的（immutable）

译注4

：

In [277]: a_tuple = (3, 5, (4, 5))In [278]: a_tuple[1] = 'four'---TypeError Traceback (most recent call last) in <module>()----> 1 a_tuple[1] = 'four'TypeError: 'tuple' object does not support item assignment

注意，仅仅因为“可以修改某个对象”并不代表“就该那么做”。这种行为在编程中也叫做副作用（side effect）。例如，在编写一个函数时，任何副作用都应该通过该函数的文档或注释明确地告知用户。即使可以使用可变对象，我也建议尽量避免副作用且注重不变性（immutability）。

标量类型

Python有一些用于处理数值数据、字符串、布尔值（True或False）以及日期/时间的内置类型。表A-2列出了主要的标量类型。后面我们将单独讨论日期/时间的处理，因为它们是由标准库中的datetime模块提供的。

表A-2：标准的Python标量类型

类型	说明
None	Python的“null”值（None只存在一个实例对象）
str	字符串类型。Python 2.x中只有ASCII值，而Python 3中则是Unicode
unicode	Unicode字符串类型
float	双精度（64位）浮点数。注意，这里没有专门的double类型
bool	True或False
int	有符号整数，其最大值由平台决定
long	任意精度的有符号整数。大的int值会被自动转换为long

数值类型

用于表示数字的主要Python类型是int和float。能被保存为int的整数的大小由平台决定（是32位还是64位），但是Python会自动将非常大的整数转换为long，它可以存储任意大小的整数。

In [279]: ival = 17239871In [280]: ival ** 6Out[280]: 26254519291092456596965462913230729701102721L

浮点数会被表示为Python的float类型。浮点数会被保存为一个双精度（64位）值。它们也可以用科学计数法表示：

In [281]: fval = 7.243In [282]: fval2 = 6.78e-5

在Python 3中，整数除法除不尽时就会产生一个浮点数：

In [284]: 3 / 2Out[284]: 1.5

在Python 2.7及以下版本中（某些读者现在用的可能就是

译注5

），只要将下面这条怪模怪样的语句添加到自定义模块的顶部即可修改这个默认行为：

from **future** import division

如果没加这句的话，你也可以显式地将分母转换成浮点数

译注6

：

In [285]: 3 / float(2)Out[285]: 1.5

要得到C风格的整数除法（如果除不尽，就丢弃小数部分），使用除后圆整运算符（//）即可：

In [286]: 3 // 2Out[286]: 1

复数的虚部是用j表示的：

In [287]: cval = 1 + 2jIn [288]: cval * (1 - 2j)Out[288]: (5+0j)

字符串

很多人都是因为Python强大而灵活的字符串处理能力才使用它的。编写字符串字面量时，既可以用单引号（'）也可以用双引号（"）：

a = 'one way of writing a string'b = "another way"

对于带有换行符的多行字符串，可以使用三重引号（即'''或"""）：

c = """This is a longer string thatspans multiple lines"""

Python字符串是不可变的。要修改字符串就只能创建一个新的：

In [289]: a = 'this is a string'In [290]: a[10] = 'f'---TypeError Traceback (most recent call last) in <module>()----> 1 a[10] = 'f'TypeError: 'str' object does not support item assignmentIn [291]: b = a.replace('string', 'longer string')In [292]: bOut[292]: 'this is a longer string'

许多Python对象都可以用str函数转换为字符串：

In [293]: a = 5.6 In [294]: s = str(a)In [295]: sOut[295]: '5.6'

由于字符串其实是一串字符序列，因此可以被当做某种序列类型（如列表、元组等）进行处理：

In [296]: s = 'python' In [297]: list(s) Out[297]: ['p', 'y', 't', 'h', 'o', 'n']In [298]: s[:3]Out[298]: 'pyt'

反斜杠（\）是转义符（escape character），也就是说，它可用于指定特殊字符（比如新行\n或unicode字符）。要编写带有反斜杠的字符串字面量，也需要对其进行转义：

In [299]: s = '12\\34'In [300]: print s12\34

如果字符串带有很多反斜杠且没有特殊字符，你就会发现这个办法很容易让人抓狂。幸运的是，你可以在字符串最左边引号的前面加上r，它表示所有字符应该按照原本的样子进行解释：

In [301]: s = r'this\has\no\special\characters'In [302]: sOut[302]: 'this\\has\\no\\special\\characters'

将两个字符串加起来会产生一个新字符串：

In [303]: a = 'this is the first half 'In [304]: b = 'and this is the second half'In [305]: a + bOut[305]: 'this is the first half and this is the second half'

字符串格式化是另一个重要的主题。Python 3带来了一些新的字符串格式化手段，这里我简要说明一下其主要机制。以一个%开头且后面跟着一个或多个格式字符的字符串是需要插入值的目标（这非常类似于C语言中的printf函数）。看看下面这个字符串：

In [306]: template = '%.2f %s are worth $%d'

在这个字符串中，%s表示将参数格式化为字符串，%.2f表示一个带有2位小数的数字，%d表示一个整数。要用实参替换这些格式化形参，需要用到二元运算符%以及由值组成的元组：

In [307]: template % (4.5560, 'Argentine Pesos', 1)Out[307]: '4.56 Argentine Pesos are worth $1'

字符串格式化是一个很大的主题，控制值在结果字符串中的格式化效果的方式非常多。我建议你在网上多找一些有关于此的资料来看看。

这里之所以要专门讨论通用字符串处理，是因为它有关于数据分析，更多细节请参阅第7章。

布尔值

Python中的两个布尔值分别写作True和False。比较运算和条件表达式都会产生True或False。布尔值可以用and和or关键字进行连接：

In [308]: True and TrueOut[308]: TrueIn [309]: False or TrueOut[309]: True

几乎所有内置的Python类型以及任何定义了**nonzero**魔术方法的类都能在if语句中被解释为True或False：

In [310]: a = [1, 2, 3] ...: if a: ...: print 'I found something!' ...:I found something!In [311]: b = [] ...: if not b: ...: print 'Empty!' ...:Empty!

Python中大部分对象都有真假的概念。比如说，如果空序列（列表、字典、元组等）用于控制流（就像上面的空列表b）就会被当做False处理。要想知道某个对象究竟会被强制转换成哪个布尔值，使用bool函数即可：

In [312]: bool([]), bool([1, 2, 3])Out[312]: (False, True)In [313]: bool('Hello world!'), bool('')Out[313]: (True, False)In [314]: bool(0), bool(1)Out[314]: (False, True)

类型转换

str、bool、int以及float等类型也可用作将值转换成该类型的函数：

In [315]: s = '3.14159'In [316]: fval = float(s) In [317]: type(fval) Out[317]: floatIn [318]: int(fval) In [319]: bool(fval) In [320]: bool(0)Out[318]: 3 Out[319]: True Out[320]: False

None

None是Python的空值类型。如果一个函数没有显式地返回值，则隐式返回None。

In [321]: a = NoneIn [322]: a is NoneOut[322]: TrueIn [323]: b = 5In [324]: b is not NoneOut[324]: True

None还是函数可选参数的一种常见默认值：

def add_and_maybe_multiply(a, b, c=None): result = a + b if c is not None: result = result * c return result

我们要牢记，None不是一个保留关键字，它只是NoneType的一个实例而已。

日期和时间

Python内置的datetime模块提供了datetime、date以及time等类型。datetime类型是用得最多的，它合并了保存在date和time中的信息：

In [325]: from datetime import datetime, date, timeIn [326]: dt = datetime(2011, 10, 29, 20, 30, 21)In [327]: dt.day In [328]: dt.minuteOut[327]: 29 Out[328]: 30

给定一个datetime实例，你可以通过调用其date和time方法提取相应的date和time对象：

In [329]: dt.date() In [330]: dt.time()Out[329]: datetime.date(2011, 10, 29) Out[330]: datetime.time(20, 30, 21)

strftime方法用于将datetime格式化为字符串：

In [331]: dt.strftime('%m/%d/%Y %H:%M')Out[331]: '10/29/2011 20:30'

字符串可以通过strptime函数转换（解析）为datetime对象：

In [332]: datetime.strptime('20091031', '%Y%m%d')Out[332]: datetime.datetime(2009, 10, 31, 0, 0)

完整的格式化定义请参见表10-2。

在对时间序列数据进行聚合或分组时，可能需要替换datetime中的一些字段。例如，将分和秒字段替换为0，并产生一个新对象：

In [333]: dt.replace(minute=0, second=0)Out[333]: datetime.datetime(2011, 10, 29, 20, 0)

两个datetime对象的差会产生一个datetime.timedelta类型：

In [334]: dt2 = datetime(2011, 11, 15, 22, 30)In [335]: delta = dt2 - dtIn [336]: delta In [337]: type(delta)Out[336]: datetime.timedelta(17, 7179) Out[337]: datetime.timedelta

将一个timedelta加到一个datetime上会产生一个新的datetime：

In [338]: dtOut[338]: datetime.datetime(2011, 10, 29, 20, 30, 21)In [339]: dt + deltaOut[339]: datetime.datetime(2011, 11, 15, 22, 30)

控制流

if、elif和else

if语句是一种最常见的控制流语句类型。它用于判断一个条件，如果为True，则执行紧跟其后的代码块：

if x < 0: print 'It's negative'

一条if语句可以跟上一个或多个elif块以及一个“滴水不漏”的else块（如果所有条件都为False）：

if x < 0: print 'It's negative'elif x == 0: print 'Equal to zero'elif 0 < x < 5: print 'Positive but smaller than 5'else: print 'Positive and larger than or equal to 5'

如果任何一个条件为True，则其后的elif或else块就不会执行。对于用and或or组成的复合条件，各条件是按从左到右的顺序求值的，而且是短路型的：

In [340]: a = 5; b = 7In [341]: c = 8; d = 4In [342]: if a < b or c > d: ...: print 'Made it'Made it

在本例中，比较运算c>d是不会被计算的，因为第一个比较运算为True。

for循环

for循环用于对集合（比如列表或元组）或迭代器进行迭代。for循环的标准语法是：

for value in collection: # 对value做一些处理

continue关键字用于使for循环提前进入下一次迭代（即跳过代码块的剩余部分）。看看下面这段代码，其功能是对列表中的整数求和并跳过None值：

sequence = [1, 2, None, 4, None, 5]total = 0for value in sequence: if value is None: continue total += value

break关键字用于使for循环完全退出。下面这段代码用于对列表的元素求和，遇到5就退出：

sequence = [1, 2, 0, 4, 6, 5, 2, 1]total_until_5 = 0for value in sequence: if value == 5: break total_until_5 += value

后面我们还会看到，如果集合或迭代器的元素是序列类型（比如元组或列表），那么还可以非常方便地将这些元素拆散成for语句中的多个变量：

for a, b, c in iterator: # 做一些处理

while循环

while循环定义了一个条件和一个代码块，只要条件不为False或循环没有被break显式终止，则代码块将一直不断地执行下去：

x = 256total = 0while x > 0: if total > 500: break total += x x = x // 2

pass

pass是Python中的“空操作”语句。它可以被用在那些没有任何功能的代码块中。由于Python是根据空白符划分代码块的，所以它的存在是很有必要的：

if x < 0: print 'negative!'elif x == 0: # TODO: 在这里放点代码 passelse: print 'positive!'

在开发一个新功能时，常常会将pass用作代码中的占位符：

def f(x, y, z): # TODO: 实现这个函数! pass

异常处理

优雅地处理Python错误或异常是构建健壮程序的重要环节。在数据分析应用中，许多函数只对特定类型的输入有效。例如，Python的float函数可以将字符串转换为浮点数，但是如果输入值不正确就会产生ValueError：

In [343]: float('1.2345')Out[343]: 1.2345In [344]: float('something')---ValueError Traceback (most recent call last) in <module>()----> 1 float('something')ValueError: could not convert string to float: something

假设我们想要编写一个在出错时能优雅地返回输入参数的改进版float函数。我们可以编写一个新函数，并把对float函数的调用放在一个try/except块中：

def attempt_float(x): try: return float(x) except: return x

只有当float(x)引发异常时，except块中的代码才会被执行：

In [346]: attempt_float('1.2345')Out[346]: 1.2345In [347]: attempt_float('something')Out[347]: 'something'

你可能已经注意到了，float还可以引发ValueError以外的异常：

In [348]: float((1, 2))---TypeError Traceback (most recent call last) in <module>()----> 1 float((1, 2))TypeError: float() argument must be a string or a number

你可能只希望处理ValueError，因为TypeError（输入参数不是字符串或数值）可能意味着你的程序中存在合法性bug。要达到这个目的，在except后面加上异常类型即可：

def attempt_float(x): try: return float(x) except ValueError: return x

于是我们就有了：

In [350]: attempt_float((1, 2))---TypeError Traceback (most recent call last) in <module>()----> 1 attempt_float((1, 2)) in attempt_float(x) 1 def attempt_float(x): 2 try:----> 3 return float(x) 4 except ValueError: 5 return xTypeError: float() argument must be a string or a number

只需编写一个由异常类型组成的元组（圆括号是必需的）即可捕获多个异常：

def attempt_float(x): try: return float(x) except (TypeError, ValueError): return x

有时你可能不想处理任何异常，而只是希望有一段代码不管try块代码成功与否都能被执行。使用finally即可达到这个目的：

f = open(path, 'w')try: write_to_file(f)finally: f.close()

这里，文件句柄f始终都会被关闭。同理，你也可以让某些代码只在try块成功时执行，使用else即可：

f = open(path, 'w')try: write_to_file(f)except: print 'Failed'else: print 'Succeeded'finally: f.close()

range和xrange

range函数用于产生一组间隔平均的整数：

In [352]: range(10)Out[352]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

可以指定起始值、结束值以及步长等信息：

In [353]: range(0, 20, 2)Out[353]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

如你所见，range所产生的整数不包括末端值。range常用于按索引对序列进行迭代：

seq = [1, 2, 3, 4]for i in range(len(seq)): val = seq[i]

对于非常长的范围，建议使用xrange，其参数跟range一样，但它不会预先产生所有的值并将它们保存到列表中（可能会非常大），而是返回一个用于逐个产生整数的迭代器。下面这段代码用于对0到9999之间所有3或5的倍数的数字求和：

sum = 0for i in xrange(10000): # %是求模运算符 if x % 3 == 0 or x % 5 == 0: sum += i

注意： 在Python 3中，range始终返回迭代器，因此也就没必要使用xrange函数了。

三元表达式

Python的三元表达式（ternary expression）允许你将产生一个值的if-else块写到一行或一个表达式中。其语法如下所示：

value = true-expr if condition else false-expr

其中的true-expr和false-expr可以是任何Python表达式。它跟下面这种冗长格式的效果一样：

if condition: value = true-exprelse: value = false-expr

下面是一个具体点的例子：

In [354]: x = 5In [355]: 'Non-negative' if x >= 0 else 'Negative'Out[355]: 'Non-negative'

跟if-else块一样，只有一个表达式会被求值。虽然这可能会引诱你总是使用三元表达式去浓缩你的代码，但要意识到，如果条件以及true和false表达式非常复杂，就可能会牺牲可读性。

数据结构和序列

Python的数据结构简单而强大。精通其用法是成为专家级Python程序员的关键环节。

元组

元组（tuple）是一种一维的、定长的、不可变的Python对象序列。最简单的创建方式是一组以逗号隔开的值：

In [356]: tup = 4, 5, 6In [357]: tupOut[357]: (4, 5, 6)

在更复杂的表达式中定义元组时，常常需要用圆括号将值围起来，比如下面这个例子，它创建了一个由元组组成的元组：

In [358]: nested_tup = (4, 5, 6), (7, 8)In [359]: nested_tupOut[359]: ((4, 5, 6), (7, 8))

通过调用tuple，任何序列或迭代器都可以被转换为元组：

In [360]: tuple([4, 0, 2])Out[360]: (4, 0, 2)In [361]: tup = tuple('string')In [362]: tupOut[362]: ('s', 't', 'r', 'i', 'n', 'g')

跟大部分其他序列类型一样，元组的元素也可以通过方括号（[]）进行访问。跟C、C++、Java之类的语言一样，Python中的序列也是从0开始索引的：

In [363]: tup[0]Out[363]: 's'

虽然存储在元组中的对象本身可能是可变的，但一旦创建完毕，存放在各个插槽中的对象就不能再被修改了：

In [364]: tup = tuple(['foo', [1, 2], True])In [365]: tup[2] = False---TypeError Traceback (most recent call last) in <module>()----> 1 tup[2] = FalseTypeError: 'tuple' object does not support item assignment# 不过In [366]: tup[1].append(3)In [367]: tupOut[367]: ('foo', [1, 2, 3], True)

元组可以通过加号（+）运算符连接起来以产生更长的元组：

In [368]: (4, None, 'foo') + (6, 0) + ('bar',)Out[368]: (4, None, 'foo', 6, 0, 'bar')

跟列表一样，对一个元组乘以一个整数，相当于是连接该元组的多个副本。

In [369]: ('foo', 'bar') * 4Out[369]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')

注意，对象本身是不会被复制的，这里涉及的只是它们的引用而已。

元组拆包

如果对元组型变量表达式进行赋值，Python就会尝试将等号右侧的值进行拆包（unpacking）：

In [370]: tup = (4, 5, 6)In [371]: a, b, c = tupIn [372]: bOut[372]: 5

即使是嵌套元组也能被拆包：

In [373]: tup = 4, 5, (6, 7)In [374]: a, b, (c, d) = tupIn [375]: dOut[375]: 7

利用该功能可以非常轻松地交换变量名。这个任务在其他许多语言中可能是下面这个样子：

tmp = aa = bb = tmpb, a = a, b

变量拆包功能常用于对由元组或列表组成的序列进行迭代：

seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]for a, b, c in seq: pass

另一个常见用法是处理从函数中返回的多个值。稍后将详细介绍。

元组方法

由于元组的大小和内存不能被修改，所以其实例方法很少。最有用的是count（对列表也是如此），它用于计算指定值的出现次数：

In [376]: a = (1, 2, 2, 2, 3, 4, 2)In [377]: a.count(2)Out[377]: 4

列表

跟元组相比，列表（list）是变长的，而且其内容也是可以修改的。它可以通过方括号（[]）或list函数进行定义：

In [378]: a_list = [2, 3, 7, None]In [379]: tup = ('foo', 'bar', 'baz')In [380]: b_list = list(tup) In [381]: b_list Out[381]: ['foo', 'bar', 'baz']In [382]: b_list[1] = 'peekaboo' In [383]: b_list Out[383]: ['foo', 'peekaboo', 'baz']

列表和元组在语义上是差不多的，都是一维序列，因此它们在许多函数中是可以互换的。

添加和移除元素

通过append方法，可以将元素添加到列表的末尾：

In [384]: b_list.append('dwarf')In [385]: b_listOut[385]: ['foo', 'peekaboo', 'baz', 'dwarf']

利用insert可以将元素插入到列表的指定位置：

In [386]: b_list.insert(1, 'red')In [387]: b_listOut[387]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']

警告： insert的计算量要比append大，因为后续的引用必须被移动以便为新元素腾地方。

insert的逆运算是pop，它用于移除并返回指定索引处的元素：

In [388]: b_list.pop(2)Out[388]: 'peekaboo'In [389]: b_listOut[389]: ['foo', 'red', 'baz', 'dwarf']

remove用于按值删除元素，它找到第一个符合要求的值然后将其从列表中删除：

In [390]: b_list.append('foo')In [391]: b_list.remove('foo')In [392]: b_listOut[392]: ['red', 'baz', 'dwarf', 'foo']

如果不考虑（使用append和remove时的）性能，Python列表可以是一种非常不错的“多重集合”数据结构。

通过in关键字，你可以判断列表中是否含有某个值：

In [393]: 'dwarf' in b_listOut[393]: True

注意，判断列表是否含有某个值的操作比字典（dict）和集合（set）慢得多，因为Python会对列表中的值进行线性扫描，而另外两个（基于哈希表）则可以瞬间完成判断。

合并列表

跟元组一样，用加号（+）将两个列表加起来即可实现合并：

In [394]: [4, None, 'foo'] + [7, 8, (2, 3)]Out[394]: [4, None, 'foo', 7, 8, (2, 3)]

对于一个已定义的列表，可以用extend方法一次性添加多个元素：

In [395]: x = [4, None, 'foo']In [396]: x.extend([7, 8, (2, 3)])In [397]: xOut[397]: [4, None, 'foo', 7, 8, (2, 3)]

注意，列表的合并是一种相当费资源的操作，因为必须创建一个新列表并将所有对象复制过去。而用extend将元素附加到现有列表（尤其是在构建一个大列表时）就会好很多。因此，

everything = []for chunk in list_of_lists: everything.extend(chunk)

要比等价的合并操作快得多

everything = []for chunk in list_of_lists: everything = everything + chunk

排序

调用列表的sort方法可以实现就地排序（无需创建新对象）：

In [398]: a = [7, 2, 5, 1, 3]In [399]: a.sort()In [400]: aOut[400]: [1, 2, 3, 5, 7]

sort有几个很不错的选项。一个是次要排序键，即一个能够产生可用于排序的值的函数。例如，我们可以通过长度对一组字符串进行排序：

In [401]: b = ['saw', 'small', 'He', 'foxes', 'six']In [402]: b.sort(key=len)In [403]: bOut[403]: ['He', 'saw', 'six', 'small', 'foxes']

二分搜索及维护有序列表

内置的bisect模块实现了二分查找以及对有序列表的插入操作。bisect.bisect可以找出新元素应该被插入到哪个位置才能保持原列表的有序性，而bisect.insort则确实地将新元素插入到那个位置上去：

In [404]: import bisectIn [405]: c = [1, 2, 2, 2, 3, 4, 7]In [406]: bisect.bisect(c, 2)Out[406]: 4In [407]: bisect.bisect(c, 5)Out[407]: 6In [408]: bisect.insort(c, 6)In [409]: cOut[409]: [1, 2, 2, 2, 3, 4, 6, 7]

警告： bisect模块的函数不会判断原列表是否是有序的，因为这样做的开销太大了。因此，将它们用于无序列表虽然不会报错，但可能会导致不正确的结果。

切片

通过切片标记法，你可以选取序列类型（数组、元组、NumPy数组等）的子集，其基本形式由索引运算符（[]）以及传入其中的start:stop构成：

In [410]: seq = [7, 2, 3, 7, 5, 6, 0, 1]In [411]: seq[1:5]Out[411]: [2, 3, 7, 5]

切片还可以被赋值为一段序列：

In [412]: seq[3:4] = [6, 3]In [413]: seqOut[413]: [7, 2, 3, 6, 3, 5, 6, 0, 1]

由于start索引处的元素是被包括在内的，而stop索引处的元素是未被包括在内的，所以结果中的元素数量是stop start。

start或stop都是可以省略的，此时它们分别默认为序列的起始处和结尾处：

In [414]: seq[:5] In [415]: seq[3:]Out[414]: [7, 2, 3, 6, 3] Out[415]: [6, 3, 5, 6, 0, 1]

负数索引从序列的末尾开始切片：

In [416]: seq[-4:] In [417]: seq[-6:-2]Out[416]: [5, 6, 0, 1] Out[417]: [6, 3, 5, 6]

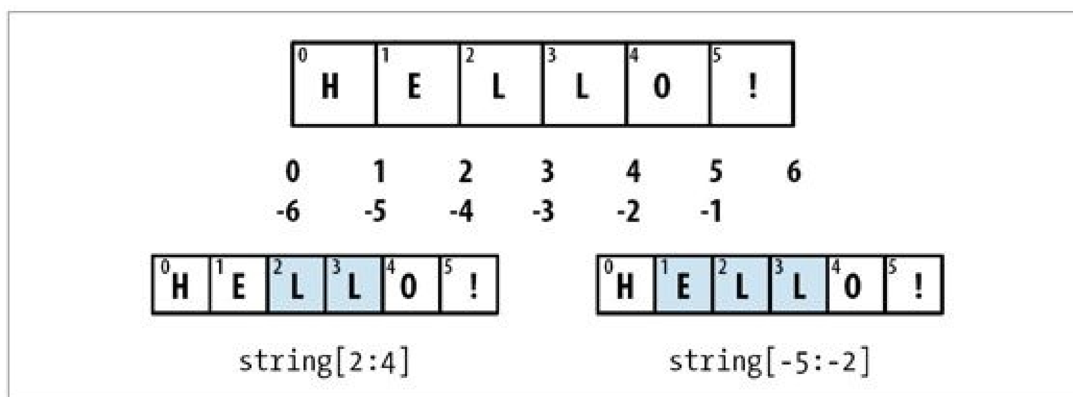
切片的语法需要花点时间去适应，尤其是当你原来用的是R或MATLAB时。图A-2形象地说明了正整数和负整数的切片过程。

还可以在第二个冒号后面加上步长（step）。比如每隔一位取出一个元素：

In [418]: seq[::2]Out[418]: [7, 3, 3, 6, 1]

在这里使用－1是一个很巧妙的办法，它可以实现列表或元组的反序：

In [419]: seq[::-1]Out[419]: [1, 0, 6, 5, 3, 6, 3, 2, 7]



图A-2：Python的切片方式

内置的序列函数

Python有一些很不错的序列函数，你应该熟悉它们，只要有机会就用。

enumerate

在对一个序列进行迭代时，常常需要跟踪当前项的索引。下面是一种DIY的办法：

i = 0for value in collection: # 用value做一些事情 i += 1

由于这种事情很常见，所以Python就内置了一个enumerate函数，它可以逐个返回序列的(i,value)元组：

for i, value in enumerate(collection): # 用value做一些事情

在对数据进行索引时，enumerate还有一种不错的使用模式，即求取一个将序列值（假定是唯一的）映射到其所在位置的字典。

In [420]: some_list = ['foo', 'bar', 'baz']In [421]: mapping = dict((v, i) for i, v in enumerate(some_list))In [422]: mappingOut[422]: {'bar': 1, 'baz': 2, 'foo': 0}

sorted

sorted函数可以将任何序列返回为一个新的有序列表：

In [423]: sorted([7, 1, 2, 6, 0, 3, 2])Out[423]: [0, 1, 2, 2, 3, 6, 7]In [424]: sorted('horse race')Out[424]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']

常常将sorted和set结合起来使用以得到一个由序列中的唯一元素组成的有序列表：

In [425]: sorted(set('this is just some string'))Out[425]: [' ', 'e', 'g', 'h', 'i', 'j', 'm', 'n', 'o', 'r', 's', 't', 'u']

zip

zip用于将多个序列（列表、元组等）中的元素“配对”，从而产生一个新的元组列表：

In [426]: seq1 = ['foo', 'bar', 'baz']In [427]: seq2 = ['one', 'two', 'three']In [428]: zip(seq1, seq2)Out[428]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]

zip可以接受任意数量的序列，最终得到的元组数量由最短的序列决定：

In [429]: seq3 = [False, True]In [430]: zip(seq1, seq2, seq3)Out[430]: [('foo', 'one', False), ('bar', 'two', True)]

zip最常见的用法是同时迭代多个序列，还可以结合enumerate一起使用：

In [431]: for i, (a, b) in enumerate(zip(seq1, seq2)): ...: print('%d: %s, %s' % (i, a, b)) ...:0: foo, one1: bar, two2: baz, three

对于“已压缩的”（zipped）序列，zip还有一个很巧妙的用法，即对该序列进行“解压”（unzip）。其实就是将一组行转换为一组列。其语法看起来有点神秘：

In [432]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'), ...: ('Schilling', 'Curt')]In [433]: first_names, last_names = zip(*pitchers)In [434]: first_namesOut[434]: ('Nolan', 'Roger', 'Schilling')In [435]: last_namesOut[435]: ('Ryan', 'Clemens', 'Curt')

稍后我将详细讨论函数调用中星号（*）的用法。其实它相当于：

zip(seq[0], seq[1], ..., seq[len(seq) - 1])

reversed

reversed用于按逆序迭代序列中的元素：

In [436]: list(reversed(range(10)))Out[436]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

字典

字典（dict）可算是Python中最重要的内置数据结构。它更常见的名字是哈希映射（hash map）或相联数组（associative array）。它是一种大小可变的键值对集，其中的键（key）和值（value）都是Python对象。创建字典的方式之一是：使用大括号（{}）并用冒号分隔键和值。

In [437]: empty_dict = {}In [438]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}In [439]: d1Out[439]: {'a': 'some value', 'b': [1, 2, 3, 4]}

访问（以及插入、设置）元素的语法跟列表和元组是一样的：

In [440]: d1[7] = 'an integer'In [441]: d1Out[441]: {7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}In [442]: d1['b']Out[442]: [1, 2, 3, 4]

你可以判断字典中是否存在某个键，其语法跟在列表和元组中判断是否存在某个值是一样的：

In [443]: 'b' in d1Out[443]: True

使用del关键字或pop方法（删除指定值之后将其返回）可以删除值：

In [444]: d1[5] = 'some value'In [445]: d1['dummy'] = 'another value'In [446]: del d1[5]In [447]: ret = d1.pop('dummy')In [448]: retOut[448]: 'another value'

keys和values方法分别用于获取键和值的列表。虽然键值对没有特定的顺序，但这两个函数会以相同的顺序输出键和值：

In [449]: d1.keys() In [450]: d1.values()Out[449]: ['a', 'b', 7] Out[450]: ['some value', [1, 2, 3, 4], 'an integer']

警告： 如果你正在使用Python 3，则dict.keys()和dict.values()会返回迭代器而不是列表。

利用update方法，一个字典可以被合并到另一个字典中去：

In [451]: d1.update({'b' : 'foo', 'c' : 12})In [452]: d1Out[452]: {7: 'an integer', 'a': 'some value', 'b': 'foo', 'c': 12}

从序列类型创建字典

有时你可能会想将两个序列中的元素两两配对地组成一个字典。粗略分析一下之后，你可能会写出这样的代码：

mapping = {}for key, value in zip(key_list, value_list): mapping[key] = value

由于字典本质上就是一个二元元组集，所以我们完全可以用dict类型函数直接处理二元元组列表：

In [453]: mapping = dict(zip(range(5), reversed(range(5))))In [454]: mappingOut[454]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}

稍后我们将讨论有关字典推导式的知识，这是构造字典的另一种优雅的方式。

默认值

下面这样的逻辑很常见：

if key in some_dict: value = some_dict[key]else: value = default_value

其实dict的get和pop方法可以接受一个可供返回的默认值，于是，上面的if-else块就可以被简单地写成：

value = some_dict.get(key, default_value)

如果key不存在，则get默认返回None，而pop则会引发一个异常。在设置值的时候，常常会将字典中的值处理成别的集类型（比如列表）。例如，根据首字母对一组单词进行分类并最终产生一个由列表组成的字典：

In [455]: words = ['apple', 'bat', 'bar', 'atom', 'book']In [456]: by_letter = {}In [457]: for word in words: ...: letter = word[0] ...: if letter not in by_letter: ...: by_letter[letter] = [word] ...: else: ...: by_letter[letter].append(word) ...:In [458]: by_letterOut[458]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}

字典的setdefault方法刚好能达到这个目的。上面的if-else块可以写成：

by_letter.setdefault(letter, []).append(word)

内置的collections模块有一个叫做defaultdict的类，它可以使该过程更简单。传入一个类型或函数（用于生成字典各插槽所使用的默认值）即可创建出一个defaultdict：

from collections import defaultdictby_letter = defaultdict(list)for word in words: by_letter[word[0]].append(word)

defaultdict的初始化器只需要一个可调用对象（例如各种函数），并不需要明确的类型。因此，如果你想要将默认值设置为4，只需传入一个能够返回4的函数即可：

counts = defaultdict(lambda: 4)

字典键的有效类型

虽然字典的值可以是任何Python对象，但键必须是不可变对象，如标量类型（整数、浮点数、字符串）或元组（元组中的所有对象也必须是不可变的）。这里的术语是可哈希性（hashability）译注7。通过hash函数，你可以判断某个对象是否是可哈希的（即可以用作字典的键）：

In [459]: hash('string')Out[459]: -9167918882415130555In [460]: hash((1, 2, (2, 3)))Out[460]: 1097636502276347782In [461]: hash((1, 2, [2, 3])) # 这里会失败，因为列表是可变的---TypeError Traceback (most recent call last) in <module>()----> 1 hash((1, 2, [2, 3])) # 这里会失败，因为列表是可变的TypeError: unhashable type: 'list'

如果要将列表当做键，最简单的办法就是将其转换成元组：

In [462]: d = {}In [463]: d[tuple([1, 2, 3])] = 5In [464]: dOut[464]: {(1, 2, 3): 5}

集合

集合（set）是由唯一元素组成的无序集。你可以将其看成是只有键而没有值的字典。集合的创建方式有二：set函数或用大括号包起来的集合字面量：

In [465]: set([2, 2, 2, 1, 3, 3])Out[465]: set([1, 2, 3])In [466]: {2, 2, 2, 1, 3, 3}Out[466]: set([1, 2, 3])

集合支持各种数学集合运算，如并、交、差以及对称差等。表A-3列出了常用的集合方法：

In [467]: a = {1, 2, 3, 4, 5}In [468]: b = {3, 4, 5, 6, 7, 8}In [469]: a | b # 并（或）Out[469]: set([1, 2, 3, 4, 5, 6, 7, 8])In [470]: a & b # 交（与）Out[470]: set([3, 4, 5])In [471]: a - b # 差Out[471]: set([1, 2])In [472]: a ^ b # 对称差（异或）Out[472]: set([1, 2, 6, 7, 8])

你还可以判断一个集合是否是另一个集合的子集（原集合包含于新集合）或超集（原集合包含新集合）：

In [473]: a_set = {1, 2, 3, 4, 5}In [474]: {1, 2, 3}.issubset(a_set)Out[474]: TrueIn [475]: a_set.issuperset({1, 2, 3})Out[475]: True

不难看出，如果两个集合的内容相等，则它们就是相等的：

In [476]: {1, 2, 3} == {3, 2, 1}Out[476]: True

表A-3：Python的集合运算

函数	其他表示法	说明
a.add(x)	N/A	将元素x添加到集合a
a.remove(x)	N/A	将元素x从集合a中删除
a.union(b)	a | b	a和b全部的唯一元素
a.intersection(b)	a & b	a和b都有的元素
a.difference(b)	a - b	a中不属于b的元素
a.symmetric_difference(b)	a ^ b	a或b中不同时属于a和b的元素
a.issubset(b)	N/A	如果a的全部元素都包含于b，则为True
a.issuperset(b)	N/A	如果b的全部元素都包含于a，则为True
a.isdisjoint(b)	N/A	如果a和b没有公共元素，则为True

列表、集合以及字典的推导式

列表推导式是最受欢迎的Python语言特性之一。它使你能够非常简洁地构造一个新列表：只需一条简洁的表达式，即可对一组元素进行过滤，并对得到的元素进行转换变形。其基本形式如下：

[expr for val in collection if condition]

这相当于下面这段for循环：

result = []for val in collection: if condition: result.append(expr)

过滤器条件可以省略，只留下表达式。例如，给定一个字符串列表，我们可以滤除长度小于等于2的字符串，并将剩下的字符串转换成大写字母形式：

In [477]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']In [478]: [x.upper() for x in strings if len(x) > 2]Out[478]: ['BAT', 'CAR', 'DOVE', 'PYTHON']

集合和字典的推导式是该思想的一种自然延伸，它们的语法差不多，只不过产生的是集合和字典而已。字典推导式的基本形式如下：

dict_comp = {key-expr : value-expr for value in collection if condition}

集合推导式跟列表推导式非常相似，唯一的区别就是它用的是花括号而不是方括号：

set_comp = {expr for value in collection if condition}

跟列表推导式一样，集合和字典的推导式也都只是语法糖而已，但它们确实能使代码变得更容易读写。再以上面那个字符串列表为例，假设我们想要构造一个集合，其内容为原列表字符串的各种长度。使用集合推导式即可轻松实现此功能：

In [479]: unique_lengths = {len(x) for x in strings}In [480]: unique_lengthsOut[480]: set([1, 2, 3, 4, 6])

再来看一个简单的字典推导式范例。我们可以为这些字符串创建一个指向其列表位置的映射关系：

In [481]: loc_mapping = {val : index for index, val in enumerate(strings)}In [482]: loc_mappingOut[482]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}

实际上，该字典还可以这样构造：

loc_mapping = dict((val, idx) for idx, val in enumerate(strings))

依我看，字典推导式版的代码要更短也更清晰。

注意： 字典和集合的推导式是最近才加入到Python的（Python 2.7和Python 3.1+）。

嵌套列表推导式

假设我们有一个由男孩名列表和女孩名列表组成的列表（即列表的列表）：

In [483]: all_data = [['Tom', 'Billy', 'Jefferson', 'Andrew', 'Wesley', 'Steven', 'Joe'], ...: ['Susie', 'Casey', 'Jill', 'Ana', 'Eva', 'Jennifer', 'Stephanie']]

这些名字可能是从多个文件中读取出来的，而且专门将男孩女孩的名字分开。现在，假设我们想要找出带有两个以上（含）字母e的名字，并将它们放入一个新列表中。我们当然可以用一个简单的for循环来实现：

names_of_interest = []for names in all_data: enough_es = [name for name in names if name.count('e') > 2]译注8 names_of_interest.extend(enough_es)

实际上，整个运算过程完全可以写成一条嵌套列表推导式，如下所示：

In [484]: result = [name for names in all_data for name in names ...: if name.count('e') >= 2]In [485]: resultOut[485]: ['Jefferson', 'Wesley', 'Steven', 'Jennifer', 'Stephanie']

乍看起来，嵌套列表推导式确实不太好理解。推导式中for的部分是按嵌套顺序排列的，而过滤条件则还是跟之前一样是放在后面的。下面是另外一个例子，将一个由整数元组构成的列表“扁平化”为一个简单的整数列表：

In [486]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]In [487]: flattened = [x for tup in some_tuples for x in tup]In [488]: flattenedOut[488]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

其实你可以这样来记：嵌套for循环中各个for的顺序是怎样的，嵌套推导式中各个for表达式的顺序就是怎样的。

flattened = []for tup in some_tuples: for x in tup: flattened.append(x)

你可以编写任意多层的嵌套，但是如果嵌套超过两三层的话，可能你就得思考一下数据结构设计有没有问题了。一定要注意上面那种语法跟“列表推导式中的列表推导式”之间的区别。比如下面这条语句也是正确的，但结果不同：

In [229]: [[x for x in tup] for tup in some_tuples]

函数

函数是Python中最主要也是最重要的代码组织和复用手段。也许并不存在拥有超级多函数的东西。实际上，我严重认为大部分程序员在做数据分析工作时所编写的函数不够多！从前面的例子中不难看出，函数是用def关键字声明的，并使用return关键字返回：

def my_function(x, y, z=1.5): if z > 1: return z * (x + y) else: return z / (x + y)

同时拥有多条return语句也是可以的。如果到达函数末尾时没有遇到任何一条return语句，则返回None。

函数可以有一些位置参数（positional）和一些关键字参数（keyword）。关键字参数通常用于指定默认值或可选参数。在上面的函数中，x和y是位置参数，而z则是关键字参数。也就是说，该函数可以下面这两种方式进行调用：

my_function(5, 6, z=0.7)my_function(3.14, 7, 3.5)

函数参数的主要限制在于：关键字参数必须位于位置参数（如果有的话）之后。你可以任何顺序指定关键字参数。也就是说，你不用死记硬背函数参数的顺序，只要记得它们的名字就可以了。

命名空间、作用域，以及局部函数

函数可以访问两种不同作用域中的变量：全局（global）和局部（local）。Python有一种更科学的用于描述变量作用域的名称，即命名空间（namespace）。任何在函数中赋值的变量默认都是被分配到局部命名空间（local namespace）中的。局部命名空间是在函数被调用时创建的，函数参数会立即填入该命名空间。在函数执行完毕之后，局部命名空间就会被销毁（会有一些例外的情况，具体请参见后面介绍闭包的那一节）。看看下面这个函数：

def func(): a = [] for i in range(5): a.append(i)

调用func()之后，首先会创建出空列表a，然后添加5个元素，最后a会在该函数退出的时候被销毁。假如我们像下面这样定义a：

a = []def func(): for i in range(5): a.append(i)

虽然可以在函数中对全局变量进行赋值操作，但是那些变量必须用global关键字声明成全局的才行：

In [489]: a = NoneIn [490]: def bind_a_variable(): ...: global a ...: a = [] ...: bind_a_variable() 译注9 ...:In [491]: print a[]

警告： 我常常建议人们不要频繁使用global关键字。因为全局变量一般是用于存放系统的某些状态的。如果你发现自己用了很多，那可能就说明得要来点儿面向对象编程了（即使用类）。

可以在任何位置进行函数声明，即使是局部函数（在外层函数被调用之后才会被动态创建出来）也是可以的：

def outer_function(x, y, z): def inner_function(a, b, c): pass pass

在上面的代码中，inner_function在outer_function被调用之前是不存在的。只要outer_function结束执行，则inner_function将会立即被销毁。

各个嵌套的内层函数可以访问其上层函数的局部命名空间，但不能绑定新变量。我将在讲解闭包的时候再对此问题进行讨论。

严格意义上来说，所有函数都是某个作用域的局部函数，这个作用域可能刚好就是模块级的作用域。

返回多个值

在我第一次用Python编程时（之前已经习惯了Java和C++），最喜欢的一个功能是：函数可以返回多个值。下面是一个简单的例子：

def f(): a = 5 b = 6 c = 7 return a, b, ca, b, c = f()

在数据分析和其他科学计算应用中，你会发现自己常常这么干，因为许多函数都可能会有多个输出（在该函数内部计算出的数据结构或其他辅助数据）。如果回忆一下本章早前讲过的元组打包和拆包功能，你可能会明白这到底是怎么一回事：该函数其实只返回了一个对象，也就是一个元组，最后该元组会被拆包到各个结果变量中。在上面的例子中，我们还可以这样写：

return_value = f()

不难看出，这里的return_value将会是一个含有3个返回值的三元元组。此外，还有一种非常具有吸引力的多值返回方式——返回字典：

def f(): a = 5 b = 6 c = 7 return {'a' : a, 'b' : b, 'c' : c}

函数亦为对象

由于Python函数都是对象，因此，在其他语言中较难表达的一些设计思想在Python中就要简单很多了。假设我们有下面这样一个字符串数组，希望对其进行一些数据清理工作并执行一堆转换：

states = [' Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlOrIda', 'south carolina##', 'West virginia?']

不管是谁，只要处理过由用户提交的调查数据，就能明白这种乱七八糟的数据是怎么一回事。为了得到一组能用于分析工作的格式统一的字符串，需要做很多事情：去除空白符、删除各种标点符号、正确的大写格式等。乍一看上去，我们可能会写出下面这样的代码：

import re # 正则表达式模块def clean_strings(strings): result = [] for value in strings: value = value.strip() value = re.sub('[!#?]', '', value) # 移除标点符号 value = value.title() result.append(value) return result

最终结果如下所示：

In [15]: clean_strings(states)Out[15]:['Alabama', 'Georgia', 'Georgia', 'Georgia', 'Florida', 'South Carolina', 'West Virginia']

其实还有另外一种不错的办法：将需要在一组给定字符串上执行的所有运算做成一个列表：

def remove_punctuation(value): return re.sub('[!#?]', '', value)clean_ops = [str.strip, remove_punctuation, str.title]def clean_strings(strings, ops): result = [] for value in strings: for function in ops: value = function(value) result.append(value) return result

然后我们就有了：

In [22]: clean_strings(states, clean_ops)Out[22]:['Alabama', 'Georgia', 'Georgia', 'Georgia', 'Florida', 'South Carolina', 'West Virginia']

这种多函数模式使你能在很高的层次上轻松修改字符串的转换方式。此时的clean_strings也更具可复用性！

还可以将函数用作其他函数的参数，比如内置的map函数，它用于在一组数据上应用一个函数：

In [23]: map(remove_punctuation, states)Out[23]:[' Alabama ', 'Georgia', 'Georgia', 'georgia', 'FlOrIda', 'south carolina', 'West virginia']

匿名（lambda）函数

Python有一种被称为匿名函数或lambda函数的东西，这其实是一种非常简单的函数：仅由单条语句组成，该语句的结果就是返回值。它们是通过lambda关键字定义的，这个关键字没有别的含义，仅仅是说“我们正在声明的是一个匿名函数”。

def short_function(x): return x *2equiv_anon = lambda x: x* 2

本书其余部分一般将其称为lambda函数。它们在数据分析工作中非常方便，因为你会发现很多数据转换函数都以函数作为参数的。直接传入lambda函数比编写完整函数声明要少输入很多字（也更清晰），甚至比将lambda函数赋值给一个变量还要少输入很多字。看看下面这个简单得有些傻的例子：

def apply_to_list(some_list, f): return [f(x) for x in some_list]ints = [4, 0, 1, 5, 6]apply_to_list(ints, lambda x: x * 2)

虽然你可以直接编写[x *2for x in ints]，但是这里我们可以非常轻松地传入一个自定义运算给apply_to_list函数。

再来看另外一个例子。假设有一组字符串，你想要根据各字符串不同字母的数量对其进行排序：

In [492]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']

这里，我们可以传入一个lambda函数到列表的sort方法：

In [493]: strings.sort(key=lambda x: len(set(list(x))))In [494]: stringsOut[494]: ['aaaa', 'foo', 'abab', 'bar', 'card']

注意：lambda函数之所以会被称为匿名函数，原因之一就是这种函数对象本身是没有提供名称属性的。

闭包：返回函数的函数

闭包（closure）不是什么很可怕的东西。如果用对了地方，它们其实可以非常强大！简而言之，闭包就是由其他函数动态生成并返回的函数。其关键性质是，被返回的函数可以访问其创建者的局部命名空间中的变量。下面是一个非常简单的例子：

```
def make_closure(a): def closure(): print('I know the secret: %d' % a) return closureclosure = make_closure(5)
```

闭包和标准Python函数之间的区别在于：即使其创建者已经执行完毕，闭包仍能继续访问其创建者的局部命名空间。因此，在上面这种情况中，返回的闭包将可打印出"I know the secret:5"。虽然闭包的内部状态（在本例中，只有值a）一般都是静态的，但也允许使用可变对象（如字典、集合、列表等可以被修改的对象）。例如，下面这个函数可以返回一个能够记录其参数（曾经传入的一切参数）的函数：

```
def make_watcher(): have_seen = {} def has_been_seen(x): if x in have_seen: return True else: have_seen[x] = True return False return has_been_seen
```

对一组整数使用该函数，可以得到：

```
In [496]: watcher = make_watcher()In [497]: vals = [5, 6, 1, 5, 1, 6, 3, 5]In [498]: [watcher(x) for x in vals]Out[498]: [False, False, False, True, True, True, False, True]
```

但是要注意一个技术限制：虽然可以修改任何内部状态对象（比如向字典添加键值对），但不能绑定外层函数作用域中的变量。一个解决办法是：修改字典或列表，而不是绑定变量。

```
def make_counter(): count = [0] def counter(): # 增加并返回当前的count count[0] += 1 return count[0]       return countercounter = make_counter()
```

你可能会想，这到底有什么用。在实际工作中，你可以编写带有大量选项的非常一般化的函数，然后再组装出更简单更专门化的函数。下面这个例子中创建了一个字符串格式化函数：

```
def format_and_pad(template, space): def formatter(x): return (template % x).rjust(space) return formatter
```

然后，你可以创建一个始终返回15位字符串的浮点数格式化器，如下所示：

```
In [500]: fmt = format_and_pad('%.4f', 15)In [501]: fmt(1.756)Out[501]: ' 1.7560'
```

如果多学一些Python面向对象编程方面的知识，你就会发现这种模式其实也能用类来实现（虽然会更啰嗦一点）。

扩展调用语法和*args*、*kwargs

在Python中，函数参数的工作方式其实很简单。当你编写func(a,b,c,d=some,e=value)时，位置和关键字参数其实分别是被打包成元组和字典的。函数实际接收到的是一个元组args和一个字典kwargs，并在内部完成如下转换：

a, b, c = argsd = kwargs.get('d', d_default_value)e = kwargs.get('e', e_default_value)

这一切都是在幕后悄悄发生的。当然，它还会执行一些错误检查，还允许你将位置参数当成关键字参数那样进行指定（即使它们在函数定义中并不是关键字参数）。

def say_hello_then_call_f(f, *args, **kwargs): print 'args is', args print 'kwargs is', kwargs print("Hello! Now I'm going to call %s" % f) return f(*args, **kwargs)def g(x, y, z=1): return (x + y) / z

然后，如果我们通过say_hello_then_call_f调用g，就会得到：

In [8]: say_hello_then_call_f(g, 1, 2, z=5.)args is (1, 2)kwargs is {'z': 5.0}Hello! Now I'm going to call Out[8]: 0.6

柯里化：部分参数应用

柯里化（currying）是一个有趣的计算机科学术语，它指的是通过“部分参数应用”（partial argument application）从现有函数派生出新函数的技术。假设我们有一个执行两数相加的简单函数：

def add_numbers(x, y): return x + y

通过这个函数，我们可以派生出一个新的只有一个参数的函数——add_five，它用于对其参数加5：

add_five = lambda y: add_numbers(5, y)

add_numbers的第二个参数称为“柯里化的”（curried）。这里没什么特别花哨的东西，因为我们其实就只是定义了一个可以调用现有函数的新函数而已。内置的functools模块可以用partial函数将此过程简化：

from functools import partialadd_five = partial(add_numbers, 5)

在讨论pandas和时间序列数据时，我们将会用该技术去创建专门的数据序列转换函数：

计算时间序列x的60日移动平均ma60 = lambda x: pandas.rolling_mean(x, 60)# 计算data中所有时间序列的60日移动平均data.apply(ma60)

生成器

能以一种一致的方式对序列进行迭代（比如列表中的对象或文件中的行）是Python的一个重要特点。这是通过一种叫做迭代器协议（iterator protocol，它是一种使对象可迭代的通用方式）的方式实现的。比如说，对字典进行迭代可以得到其所有的键：

In [502]: some_dict = {'a': 1, 'b': 2, 'c': 3}In [503]: for key in some_dict: ...: print key, 译注10a c b

当你编写for key in some_dict时，Python解释器首先会尝试从some_dict创建一个迭代器：

In [504]: dict_iterator = iter(some_dict)In [505]: dict_iteratorOut[505]:

迭代器是一种特殊对象，它可以在诸如for循环之类的上下文中向Python解释器输送对象。大部分能接受列表之类的对象的方法也都可以接受任何可迭代对象。比如min、max、sum等内置方法以及list、tuple等类型构造器：

In [506]: list(dict_iterator)Out[506]: ['a', 'c', 'b']

生成器（generator）是构造新的可迭代对象的一种简单方式。一般的函数执行之后只会返回单个值，而生成器则是以延迟的方式返回一个值序列，即每返回一个值之后暂停，直到下一个值被请求时再继续。要创建一个生成器，只需将函数中的return替换为yeild即可：

def squares(n=10): for i in xrange(1, n + 1): print 'Generating squares from 1 to %d' % (n **2)** 译注**11 yield i** 2

调用该生成器时，没有任何代码会被立即执行：

In [2]: gen = squares()In [3]: genOut[3]:

直到你从该生成器中请求元素时，它才会开始执行其代码：

In [4]: for x in gen: ...: print x, ...:Generating squares from 0 to 1001 4 9 16 25 36 49 64 81 100

假设我们希望找出“将1美元（即100美分）兑换成任意一组硬币”的所有唯一方式。你可能会想出很多种实现办法（包括“已找到的唯一组合”的保存方式）。下面我们编写一个生成器来产生这样的硬币组合（硬币面额用整数表示）：

def make_change(amount, coins=[1, 5, 10, 25], hand=None): hand = [] if hand is None else hand if amount == 0: yield hand for coin in coins: # 确保我们给出的硬币没有超过总额，且组合是唯一的 if coin > amount or (len(hand) > 0 and hand[-1] < coin): continue for result in make_change(amount - coin, coins=coins, hand=hand + [coin]): yield result

这个算法的细节并不重要（你能想出一个更短点的办法吗？）。然后我们可以编写：

In [508]: for way in make_change(100, coins=[10, 25, 50]): ...: print way[10, 10, 10, 10, 10, 10, 10, 10, 10, 10][25, 25, 10, 10, 10, 10, 10][25, 25, 25, 25][50, 10, 10, 10, 10, 10][50, 25, 25][50, 50]In [509]: len(list(make_change(100)))Out[509]: 242

生成器表达式

生成器表达式（generator expression）是构造生成器的最简单方式。生成器也有一个类似于列表、字典、集合推导式的东西，其创建方式为，把列表推导式两端的方括号改成圆括号：

In [510]: gen = (x ** 2 for x in xrange(100))In [511]: genOut[511]:

它跟下面这个冗长得多的生成器是完全等价的：

def _make_gen(): for x in xrange(100): yield x ** 2gen = _make_gen()

生成器表达式可用于任何接受生成器的Python函数：

In [512]: sum(x **2 for x in xrange(100))Out[512]: 328350In [513]: dict((i, i** 2) for i in xrange(5))Out[513]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

itertools模块

标准库itertools模块中有一组用于许多常见数据算法的生成器。例如，groupby可以接受任何序列和一个函数。它根据函数的返回值对序列中的连续元素进行分组。下面是一个例子：

In [514]: import itertoolsIn [515]: first_letter = lambda x: x[0]In [516]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']In [517]: for letter, names in itertools.groupby(names, first_letter): ...: print letter, list(names) # names是一个生成器A ['Alan', 'Adam']W ['Wes', 'Will']A ['Albert']S ['Steven']

表A-4中列出了一些我经常用到的itertools函数。

表A-4：一些常用的itertools函数

函数	说明
imap(func, *iterables)	内置函数map的生成器版，将func应用于参数序列的各个打包元组
ifilter(func, iterable)	内置函数filter的生成器版，当func(x)为True时输出元素x

表A-4：一些常用的itertools函数（续）

函数	说明
combinations(iterable, k)	生成一个由iterable中所有可能的k元元组组成的序列（不考虑顺序）
permutations(iterable, k)	生成一个由iterable中所有可能的k元元组组成的序列（考虑顺序）
groupby(iterable[, keyfunc])	为每个唯一键生成一个（key, sub-iterator）。

注意：许多在Python 2（itertools）中产生列表的内置函数（如zip、map、filter等），在Python 3中都被换成了其生成器版。

文件和操作系统

本书的代码示例大多使用诸如pandas.read_csv之类的高级工具将磁盘上的数据文件读入Python数据结构。但我们还是需要了解一些有关Python文件处理方面的基础知识。好在它本来就很简单，这也是Python在文本和文件处理方面的如此流行的原因之一。

为了打开一个文件以便读写，可以使用内置的open函数以及一个相对或绝对的文件路径：

In [518]: path = 'ch13/segismundo.txt'In [519]: f = open(path)

默认情况下，文件是以只读模式（'r'）打开的。然后，我们就可以像处理列表那样来处理这个文件句柄f了，比如对行进行迭代：

for line in f: pass

从文件中取出的行都带有完整的行结束符（EOL），因此你常常会看到下面这样的代码（得到一组没有EOL的行）：

In [520]: lines = [x.rstrip() for x in open(path)]In [521]: linesOut[521]:['Sue\xc3\xb1a el rico en su riqueza,', 'que m\xc3\xa1s cuidados le ofrece;', '', 'sue\xc3\xb1a el pobre que padece', 'su miseria y su pobreza;', '', 'sue\xc3\xb1a el que a medrar empieza,', 'sue\xc3\xb1a el que afana y pretende,', 'sue\xc3\xb1a el que agravia y ofende,', '', 'y en el mundo, en conclusi\xc3\xb3n,', 'todos sue\xc3\xb1an lo que son,', 'aunque ninguno lo entiende.', '']

如果输入f =open(path,'w')，就会有一个新文件被创建在ch13/segismundo.txt，并覆盖掉该位置原来的任何数据。表A-5列出了所有可用的文件读写模式。

表A-5：Python的文件模式

模式	说明
r	只读模式
w	只写模式。创建新文件（删除同名的任何文件[译注12]）
a	附加到现有文件（如果文件不存在则创建一个）
r+	读写模式
b	附加说明某模式用于二进制文件，即'rb'或'wb'
U	通用换行模式。单独使用'U'或附加到其他读模式（如'rU'）

译注12：这的“名”包括路径。

要将文本写入文件，可以使用该文件的write或writelines方法。例如，我们可以创建一个无空行版的prof_mod.py

译注13

，如下所示：

In [522]: with open('tmp.txt', 'w') as handle: .....: handle.writelines(x for x in open(path) if len(x) > 1)In [523]: open('tmp.txt').readlines()Out[523]:['Sue\xc3\xb1a el rico en su riqueza,\n','que m\xc3\xa1s cuidados le ofrece;\n','sue\xc3\xb1a el pobre que padece\n','su miseria y su pobreza;\n','sue\xc3\xb1a el que a medrar empieza,\n','sue\xc3\xb1a el que afana y pretende,\n','sue\xc3\xb1a el que agravia y ofende,\n','y en el mundo, en conclusi\xc3\xb3n,\n','todos sue\xc3\xb1an lo que son,\n','aunque ninguno lo entiende.\n']

表A-6列出了一些最常用的文件方法。

表A-6：重要的Python文件方法或属性

方法	说明
read([size])	以字符串形式返回文件数据，可选的size参数用于说明读取的字节数
readlines([size])	将文件返回为行列表，可选参数size
write(str)	将字符串写入文件
close()	关闭句柄
flush()	清空内部I/O缓存区，并将数据强行写回磁盘
seek(pos)	移动到指定的文件位置（整数）
tell()	以整数形式返回当前文件位置
closed	如果文件已关闭，则为True

译注1

：这里只是作者起的名字而已，不必介怀，你完全可以给它起个“真命天子类型”之类的名字。其实它是一个哲学和逻辑学概念，就是说“对于一只鸟类动物，不用管它到底是不是鸭子，只要看它像不像子就可以了”。

译注2：也就是定义别名。

译注3：在函数式编程中，也常译作惰性求值。

译注4

：这个词指的是“不能修改原内存块的数据”也就是说，即使修改操作成功了，也只是创建了一个新对象并将其引用赋值给原变量而已。

译注5：作者用的比我现在用的版本还老。所以在阅读本书的过程中有些例子的计算结果不一定跟书上的完全一致。

译注6

：分子也可以的。

译注7

：或者翻译成可散列性。

译注8：应该是">=", 因为原文是"two and more"。

译注9

：注意缩进，别搞成递归了。

译注10：注意这里的逗号。

译注11

：应该放到for循环之前，否则后面的执行结果与书上的不一样。

译注13

：应该是segismundo.txt。

Table of Contents

前言

1章　准备工作

本书主要内容

为什么要使用

Python进行数据分析

重要的Python库

安装和设置

社区和研讨会

使用本书

致谢

第2章　引言

来自bit.ly的1.usa.gov数据

MovieLens 1M数据集

1880—2010年间全美婴儿姓名

小结及展望

第3章　IPython：一种交互式计算和开发环境

IPython基础

内省

使用命令历史

与操作系统交互

软件开发工具

IPython HTML Notebook

利用IPython提高代码开发效率的几点提示

高级IPython功能

致谢

第4章　NumPy基础：数组和矢量计算

NumPy的ndarray：一种多维数组对象

通用函数：快速的元素级数组函数

利用数组进行数据处理

用于数组的文件输入输出

线性代数

随机数生成

范例：随机漫步

第5章　pandas入门

pandas的数据结构介绍

基本功能

汇总和计算描述统计

处理缺失数据

层次化索引

其他有关pandas的话题

第6章　数据加载、存储与文件格式

读写文本格式的数据

二进制数据格式

使用HTML和Web API

使用数据库

第7章　数据规整化：清理、转换、合并、重塑

合并数据集

重塑和轴向旋转

数据转换

字符串操作

示例：USDA食品数据库

第8章　绘图和可视化

matplotlib API入门

pandas中的绘图函数

绘制地图：图形化显示海地地震危机数据

Python图形化工具生态系统

第9章　数据聚合与分组运算

GroupBy技术

数据聚合

分组级运算和转换

透视表和交叉表

示例：2012联邦选举委员会数据库

第10章　时间序列

日期和时间数据类型及工具

时间序列基础

日期的范围、频率以及移动

时区处理

时期及其算术运算

重采样及频率转换

时间序列绘图

移动窗口函数

性能和内存使用方面的注意事项

第11章　金融和经济数据应用

数据规整化方面的话题

分组变换和分析

更多示例应用

第12章　NumPy高级应用

ndarray对象的内部机理

高级数组操作

广播

ufunc高级应用

结构化和记录式数组

更多有关排序的话题

NumPy的matrix类

高级数组输入输出

性能建议

附录A　Python语言精要